# DATA BASE MANAGEMENT FOR THE APPLE™

*LEARN HOW TO KEEP HOUSEHOLD ACCOUNTS*
*MANAGE INVESTMENT PORTFOLIOS*
*SCHEDULE APPOINTMENTS*
*ORGANIZE TAX RECORDS*
*SORT MAILING LISTS*
*TRACK INVENTORIES*
*AND MUCH MORE*
*USING **APPLESOFT** ROUTINES*
*THAT ARE EASY TO UNDERSTAND*
*AND EASY TO PROGRAM*
*ON THE **APPLE II PLUS** AND **APPLE IIe***

**NAT WADSWORTH**


HAYDEN

# DATA BASE MANAGEMENT FOR THE APPLE™

NAT WADSWORTH

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

Acquisitions Editor: DOUGLAS McCORMICK
Production Editor: TERRY DONOVAN
Art Director: JIM BERNARD
Cover Illustration: GEORGE BAQUERO
Original Artwork: JOHN McAUSLAND
Composition: McFARLAND GRAPHICS AND DESIGN, INC.
Printed and bound by: ARCATA GRAPHICS CO.; FAIRFIELD GRAPHICS DIVISION

**Library of Congress Cataloging in Publication Data**

Wadsworth, Nat.
Data base management for the Apple.

1. Data base management. 2. Apple computer. I. Title. QA76.9.D3W3
1983 001.64'2 83-4387
ISBN 0-8104-6282-6



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | PRINTING |
|---|---|---|---|---|---|---|---|---|---|
| 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | YEAR |

# PREFACE

The primary objectives of this book are to:

1) Serve as an introduction to using a personal computer to organize and manipulate information. In some circles this general concept is referred to loosely as data base management.

2) Present a rudimentary, but functional, information management program written in high-level language (BASIC). This program, by itself, may be used to explore and understand fundamental computerized information management processes.

3) Provide, for the more advanced student or enthusiast, a well-documented-program framework from which one may proceed to modify and enhance the capabilities of the core program.

In striving to meet such goals, a large number of program design decisions had to be made involving numerous constraints. Where conflicts arose, the basic principles followed were to opt for simplicity of operation and/or presentation at the sacrifice of programming sophistication, operating speed, or advanced capabilities.

After many years of programming, I am thoroughly convinced that no single program can come even close to satisfying all the requirements of the general public at large. By providing the type of extensive program documentation included in this book, however, I have attempted to provide a platform from which the dissatisfied can build or restructure to their heart's content. This provides considerably more freedom of choice than one receives when one purchases a program on a "locked" disk.

I believe that providing freedom of choice is a good part of what life is all about. An exciting aspect about the use of personal computers is that they can aid us in making suitable choices. An exciting aspect about data base management is that it can help us to organize information so that we might make intelligent suitable choices. An exciting concept about computer programs in general is that, if we know how they work, we can make suitable intelligent choices concerning how they perform in the future.

Nat Wadsworth

# CONTENTS

# 1 INTRODUCTION TO COMPUTERIZED DATA BASE MANAGEMENT

"Data base management." Everybody seems to be using this phrase these days. Just what does someone mean when they use such terminology? The truth of the matter is that the exact meaning of the phrase "data base management" can depend to some degree on the context in which it is used!

A person who works in the accounts receivable department of a large firm, whose job is to track and collect "past-due" accounts, is performing data base management.

An individual who keeps a personal address book of friends, relatives, and associates is performing data base management.

And a teacher who maintains records of students' grades and attendance is certainly involved with data base management.

The Internal Revenue Service (IRS) is deeply involved in many aspects of data base management: massive data base management. Ah, but let's try to keep this text cheerful . . . .

You see, the central concept of data base management—and you may verify that it applies to all of the above examples—is simply this: it is the *physical control of information.*

Why control information? So you can use it more effectively! Random collections of facts have little value. Organized and compiled knowledge can have tremendous value.

The accounts receivable clerk who merely knows that "some businesses owe money" is going to have a difficult time coming up with the cash so that the company can meet its next payroll. But the same clerk who culls the files to identify the accounts that are indebted, who extracts the invoice numbers, amounts due, and who obtains the telephone numbers of those accounts is in a position to take immediate action. Action that can result in those bills being paid promptly!

If you keep the names and addresses of all your friends, relatives, and acquaintances on scraps of paper tucked in various nooks and crannies around your home, you are probably going to lose track of some

associates. But if you organize that information in, say, a little old address book and you organize that address book by your contacts' last names, then, oh yes, you can quickly look up the phone number or address of those with whom you desire to communicate.

Likewise, lots of luck to the teacher who didn't keep meticulous grade records so that the progress of students could be reviewed with parents, other teachers, and the pupils themselves.

As for the IRS, wouldn't it be a shame if they suddenly lost the ability to organize and control all that tax information? It could certainly result in chaos!

### Organizing Is a Lot of Work

Sure it is. That is one of the big problems with managing information. It takes a lot of hard work and effort to keep it organized.

There is a universal physical law often referred to by scientists, known as the Law of Entropy. The gist of this theory is that, if left to themselves, things tend to fall apart or go into disarray. Well, this physical law wasn't intended to serve as a theorem in information management. But if you look at your own personal experience, you may tend to agree that there sure seems to be some kind of "force" at work that applies to keeping track of information. If left unattended, information seems to dissipate and become increasingly disorganized! It seems to take lots of work—energy—to keep information arranged in usable form.

Trouble is, people seem to have an aversion to constantly struggling to keep information organized. Ever notice how people in an office would generally prefer to go to a meeting or talk on the phone rather than straighten out that pile of papers on their desks?

Do you really *enjoy* periodically having to go through your little old address book (particularly during the holiday season when you are supposed to be sending cards, but you would prefer to be going to parties)? With 25% of the population moving every year, it amounts to a lot of messy erasing and updating just to keep a little old address book in order.

I have been told by practicing teachers that keeping a constantly updated record of students' attendance and grades is a lot of work. It sure doesn't seem to be a lot of fun, from what I have been able to observe at close hand.

The good old IRS employs thousands of people just to keep their information about how you pay your taxes organized. They pay those people with lots of *your* money. So, not only is organizing information a lot of work, it is expensive!

## Let the Computer Do It

The work, that is. Sure. That is what this book is all about. Let's have a computer do as much of the work as possible when it comes to organizing and compiling information. Then you and I can have more time to take advantage of that information and put it to good use.

Another way of looking at information is to consider it as *power*! It takes energy to acquire, organize, and interpret data. It takes effort to keep it ordered. But, once harnessed and controlled, information can produce concrete results. Results equals personal power.

Of course, you know that a computer is well suited for performing many of the tasks associated with managing different kinds of information. It can be programmed to be proficient at storing, retrieving, arranging, rearranging, and extracting specific facts and figures. Frequently, it can perform such operations in a blink of the


**Fig. 1-1. Your Apple II computer can quickly organize data and output specific information by using the data base program in Chap. 3.**

proverbial eye. Even large tasks that might take a person several days—such as sorting operations—can be executed in a matter of minutes when performed by a computer such as the Apple II, Apple II Plus, or Apple IIe.

The computer has other advantages over a person when it comes to performing rote work. It is accurate. Unless the computer is malfunctioning or the program it is executing (heaven forbid!) has flaws, it will unerringly adhere to the rules laid down for its performance. There are no slip-ups, no misclassifications, no misfilings into the wrong bin where the data might be lost forever. It does precisely (and only) what it is told to do. (Of course, if *you* tell it to do the wrong thing, it will do that too!)

It can handle large volumes of information. Quantities of information that would reduce most people to little mounds of total frustration are routinely processed by the machine without so much as a whimper.

The computer can operate unattended. Once you have told it what is to be accomplished, you can often walk away from the machine while it cranks away on its own. Need a mailing list in zip code order? Tell the computer to print it out while you go to lunch!

Are you convinced? Sure you are. You know your Apple can save you a lot of time and trouble by processing and organizing data. Up until now, you just may not have known how to tell your computer how to do it! That obstacle is about to be eliminated.

# 2 TOWARD UNDERSTANDING DATA BASE MANAGEMENT

Just knowing that a computer, equipped with the proper program, can perform a stated task, is not enough in itself to get you very far. Like all tasks one is likely to encounter, there is often more to know about the process than simple recognition that the job can be accomplished.

So it is, indeed, with data base management. It is not enough to merely be aware of the fact that a computer can be programmed to manipulate data in a variety of ways. While it is a good starting point to know that you can extract particular kinds of information, analyze large amounts of information to uncover hidden relationships and correlations or sort data into convenient groups, it is not enough to actually accomplish real objectives.

No, in order to effectively use a computer to manage data, you must have at least an elementary appreciation of how it goes about accomplishing its work. And, in order to begin discussing that subject, it is necessary to know something about how information can be processed. That is, how it can be broken down into its component parts so that it can be accessed and physically manipulated by a computer.

## A Data Base Vocabulary

It will behoove us to define some fundamental terms at this point so that you can understand what it is I am talking about as I discuss various aspects of data base management.

I like to start at the ground level and work up when exploring new territory. Thus, let's start by defining the smallest piece of information that can be dealt with by a computerized information management system.

It is a *character.* That is, a single letter (of the alphabet), numeric digit (Ø-9), punctuation mark, special symbol, or control code (such as a carriage return, linefeed, or the "escape" indicator).

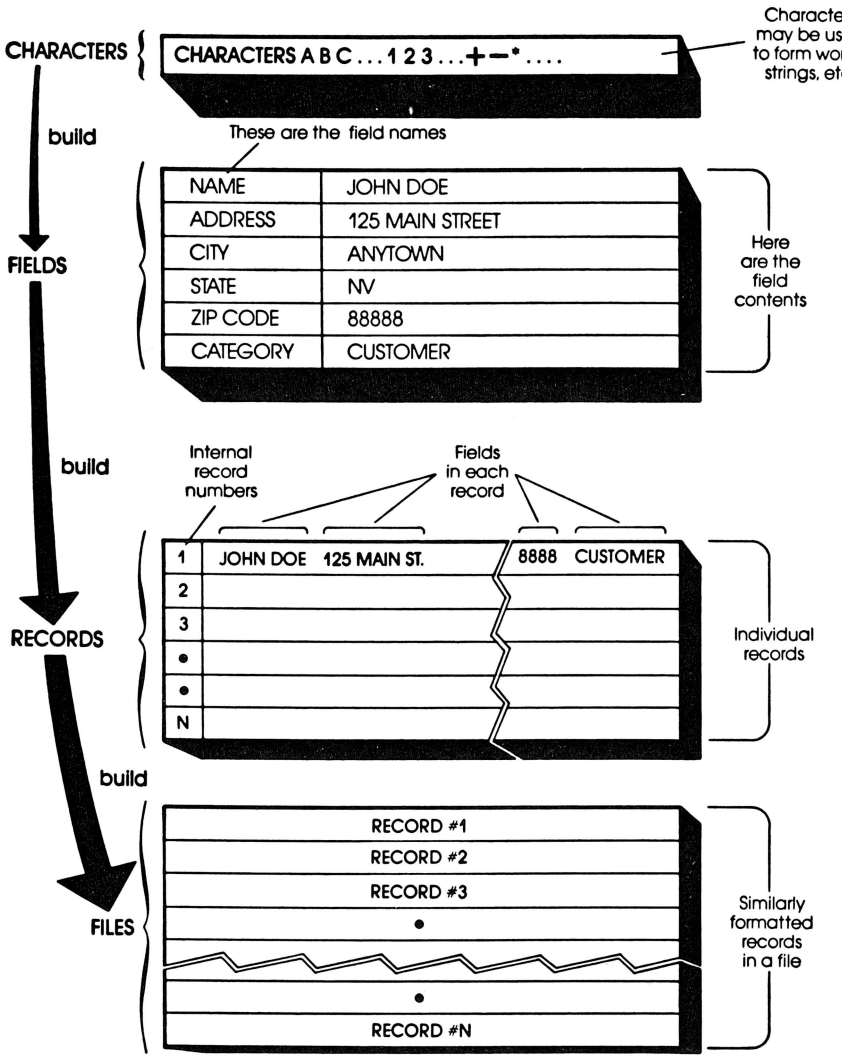


**Fig. 2-1. Building a data base: characters make up fields; fields make up records; and records make up files.**

Now a computer actually processes characters in encoded form. In reality, each character is broken down still further, into binary bits. But, as far as we will be concerned at the practical level, the decoding of individual characters into binary patterns—such as ASCII code—will be considered transparent to the user. Such encoding is taken care of by the machinery without intervention on our part. All we need know is that, when we press a key on a keyboard, the computer is going to interpret it as representing a unique character. This is the smallest unit of

information that will be inputted or received as output from the data base management program.

If you put a bunch of characters together you have what is sometimes referred to by computer technologists as a character *string.* The term "string" in this context refers to a group of characters that are (or may) be manipulated as an entity. The concept is important because it is often used to define the particular group of characters that a computer is to look for during a search operation. Note that a string may actually consist of just a few characters (such as might make up part of a word) or a large number of characters (such as might be encompassed by several words or a phrase). (In fact, a string might also be singular, in that it is represented only by a single character.)

A *word* can be defined as a string of characters separated by a space. (Remember, as far as a computer is concerned, a space, such as occurs between each word on this page, is actually represented by a specific character: the *space* character!) People are quite familiar with the concept of words. Computers have to be taught how to recognize words. In the context of data base management, a space means the end of a word to the computer!

Now we come to a very important concept in data base management: a *field.* A field can best be thought of as a logical element. That is, the information within a field is generally considered to relate to a specific purpose or function. The contents of a field may consist of entire strings, words, or just individual characters.

Some examples of "fields" may help illustrate the concept. The first name, middle initial, and last name of a person might all be placed together in a field. This field might be referred to as a "name" field, because it contains all the information pertaining to an individual's name.

All of the information relating to a person's or a firm's street address, such as the house or building number, the name of the street, boulevard or avenue, and even the apartment or suite number, could be grouped together in a field. Such a field might be logically referred to as the "local address" field.

Note that in these fields there are a number of words, but they all relate to the same logical subject, such as local address or the name of a person.

The concept of a field is important in data base management applications. This is because certain operations, such as search and sort procedures, are performed relative to the contents of specific fields. Once fields have been defined for a particular data base, most of the manipulative power provided by the program will focus at the field level. It is thus important to fully appreciate the divisional effects of field boundaries.

While fields typically contain information that is logically or symbolically related to other information within the same field, this does

not *have* to be the case. Physically, a field in the data base program provided in this book is delineated by the inputting of a carriage return (end of line) character.

Furthermore, the maximum size (length) of any field in this data base management system is limited to a total of 40 characters (including spaces, punctuation, special symbols, etc.). And, as will be pointed out shortly, there is a limit to the maximum number of fields that may be defined for a particular data base.

Finally, before going on to other definitions, it will be pointed out that, in this system, there can be two different *types* of data fields.

*Alphanumeric* fields can contain any valid characters, such as letters of the alphabet, digits, and special symbols. These characters may be used without regard to format within a field. The alphanumeric field is the most commonly used type of field.

*Numeric* fields may only contain a single numeric value, expressed precisely in terms of significant digits, possibly including a sign and a decimal point. The significance and application of numeric fields will be discussed later.

One or more fields may be organized in a fixed sequential relationship to form a *record.* A record normally forms a logically complete structure. That is, in an application such as maintaining a mailing list, a record might typically hold an individual's name, street address, city and state, zip code, telephone number, and a classification code. Each of those separately identified parts of the record would comprise a field. Taken together, those fields constitute a record. Note that all of the information within each record relates to a particular individual.

Records are important in the data base system being presented here. Records are the smallest physical units that can be created, removed, or moved about within a data file.

Which leads us to the definition of a file! A data*file* is a collection of records, all of which have a specific, defined format. Typically, all records in a file relate to a particular application, such as maintaining a mailing list.

In this data base management system, all of the records in a single file are stored in the computer's memory at one time. That is, the file is considered to be "memory-resident" whenever it is being processed by the computer. Of course, when not in use, the contents of a file can be stored on a diskette for safekeeping.

A *data base* may be considered as consisting of all of the information that you have, organized in a form that can be processed by your computer system. As you create new files of information on your computer system, your data base will expand to give you more and more personal information-handling power!

## Controlling Your Data Base

The program that directs the operation of your computerized data base management system works in two fundamental modes. One mode is referred to as the *data entry* mode. When in this mode, information is accepted as data to be stored and manipulated by the computer.

The second mode is termed the *command* mode. Commands are the directives you give to the program to cause it to act on the information in a data file in a prescribed manner. These commands provide the ability to add data records to a file, change data in selected records, and insert or delete records. Other commands permit you to search for specific information by examining the contents of fields and to arrange the records in alphabetical order as a function of the contents of a specified field. There is even a directive that permits the summing of arithmetic values in numeric fields.

Still other commands enable entire data files to be stored on a diskette or transferred from a diskette into memory.

The program commands give you the ability to personally control every aspect of your new information-processing tool!

You may have noticed that the types of commands mentioned above can be grouped into several categories. One category might be viewed as having to do with the *editing* of information in a data file. The ability to add or delete information in a file or to change the contents of all or part of a record allows you to conveniently keep your information up to date. For instance, if you are maintaining a mailing list, you need the ability to enter a person's change of address in the specific record that pertains to that individual.

Search and sort commands provide for the *organizing* of information. The search directive screens out unwanted information and displays only those records containing sought-after data. For instance, if you wanted to locate all of the people in a mailing list that lived in a particular state, you could direct the data base program to search all the data in a file and display just those records having an address in the desired state.

The sort directive permits you to group information into categories, and even subcategories. If you wanted to take advantage of third class mailing rates at the post office, you would have to present all your mail arranged according to zip codes around the country. This is a difficult task to perform by hand. However, the computer can accomplish it with ease by merely arranging the contents of a file on the basis of a zip code field!

A third type of command may be classified in the *mathematical* category. The ability to "tally" the numbers in a specified field for a group of records has considerable value in many applications. For

instance, the individual prices of components in a complex system could quickly be summed by the computer to arrive at a final system cost.

And finally, there are the directives related to data *storage.* These are the commands used to place a data file on a diskette or to reactivate a file by bringing it back into memory from a diskette.

### Formatting Your Data

Now, the key to successfully utilizing a data base management program lies in preparing a good format for your raw data, that is, in arranging your initial information so that it can be effectively processed and analyzed by the computer. A good portion of this book will be devoted to providing practical examples of how data can be formatted to accomplish various applications.

Right now, it will help you get a feel for the process of initially formatting a data base by reviewing the following:

Data within a file is organized as records. Each record is assigned (by the computer program) a reference number. This reference number is maintained without the operator's assistance, but it may be referred to by the operator to perform certain operations. Thus, the first record inputted into a file is given the internal reference number one. The next record is referenced as record number two, and so forth. If the operator wants to remove a certain record from the file, the number of that record can be referenced and the deletion accomplished. After a deletion, the records remaining in the file would automatically be assigned numbers to reflect their new positions within the file. Similarly, if a sorting operation rearranged the order of records in a file, the number assigned to each record might be altered. Again, all of this is taken care of by the computer program. The internal numbering system is just a means of communicating to the user the actual physical location of each record within a file at any given time.

You know already that each record is further subdivided into fields. A field is usually assigned a specific function or purpose (though this does not have to be the case). The maximum number of characters in a field is decided by the user for each application (with an upper limitation of 40 characters to a field). Also, fields are designated as alphanumeric or strictly numeric, depending on the type of data that is to be stored. All of the information in an individual field in this data base management system will normally appear on a single line of the display.

The concept of a field is important because each field delineates a boundary within a record. Furthermore, the data base management program operates on the contents of fields. That is, the data within records is processed on a field-by-field basis. To assist the data base operator, fields may be assigned reference names when a file is initially formatted. Thus, if one of the fields in a record will always contain a

person's phone number, the field might be assigned "PHONE NR" as a name. Thus, each time that portion of a record was accessed by a user, the field name would be presented as a reminder of the type of data contained therein.

Fields contain characters arranged as words or strings. There is a limit to the number of fields and the total number of characters that can be assigned to any record. (These specifications will be detailed later in this manual.)

Remember, a character is the smallest amount of information or data that can be manipulated by the data base operator in this system.

## A Few Conceptual Examples

With the background you now have, it is time to present a few simple illustrations of how a data base management program may be put to practical use.

Let us present one example by assuming that you are a salesperson for an insurance company. You work out of an office in your home on an essentially independent, unsupervised basis. You are basically an industrious person, constantly striving to improve your performance (primarily because that increases your income).

Your company creates leads for you by advertising in various magazines. People who respond to the ads fill out a coupon, indicating their interest and providing their name and address. Your company forwards batches of these coupons to you each week. A big part of your job as a salesperson is to follow up on these leads and try (very hard) to sell these people suitable insurance policies. Of course, the company you work for carries many lines of insurance, so you have clients interested in many different types of policies.

Not only must you constantly strive to obtain new customers, you must also continue to service your growing base of regular customers. Periodically, you need to call on them, review their current needs, update policies, and so forth.

You are, indeed, becoming a very busy person. You come to the realization that you are starting to face a real problem just keeping all your clients and prospects organized. You have, for some years, maintained a large notebook. You have even gone so far as to devise a system whereby you create a new page for each customer you sell. You file these pages in alphabetical order. By making pertinent notes on these pages you (usually) are able to keep some kind of a handle on the situation. You do this by periodically scanning all these notes, keeping an eye out for policies that are about to lapse, news of an increase in the size of a family, changes in car registration, etc. These are clues that the customer should be contacted for policy updating.

If that wasn't enough to keep you busy, the real problem centers around those ever-arriving new prospect coupons. Your present system is to simply stuff each pile of new coupons into your briefcase. As you get time, you take out a few coupons, try to make contact with the prospect and set up an appointment. You admit to yourself that the system is a bit messy and you have no doubt that a few prospects "fall through the cracks." It is definitely time to do something about the matter.

Fortunately, you have an Apple II computer just sitting around your home office with nothing to do and you have wisely obtained a copy of this book. Now, let's look at what you could do in this particular situation.

You could establish a data file to keep track of your customers and prospects (and even your friends and associates)! For illustrative purposes, we will make the structure of this file very simple. But you will soon realize that you could increase the power of this information management system by expanding the number of fields employed in a record.

Let us say that you define each record in the file to consist of six fields. You name the fields as follows: Name, Address, City, State, Zip Code, and Category. (The names you assign serve to identify the contents of each field.)

You establish your data base by sitting down one day and entering all of the relevant information into the computer. The information is simply transferred from those coupons the company sends you (that are filling your briefcase).

You decide to categorize each prospect on the basis of the type of insurance that person has expressed an interest in, such as automobile, life, property, or medical coverage. This information is entered in the "Category" field.

Perhaps you end up with several hundred names and addresses in your initial data base. Of course, information has been entered essentially at random as you grabbed inquiry coupons out of your briefcase. You decide the first thing you would like to do is have your list of prospects arranged in zip code order. You press a few keys on the keyboard of your computer. The computer arranges all the records in the file into zip code order in accordance with the contents of the "Zip Code" field. You press a few more keys and the system outputs the ordered list on your printer.

Next you decide you could become more effective at your selling efforts by "softening up" potential clients with an introductory letter. What is more, you want to slant the letter depending on the type of insurance with which the prospect has expressed an interest. So, you sort your data file on the basis of the contents of the "Category" field. This groups all the prospects according to interest. After this operation you find that record numbers 1 to 40 contain clients interested in automobile

coverage, 41 to 55 in life, 56 to 8Ø in medical, and numbers 81 to 18Ø are looking for property coverage. Wow! You have just discovered something you never fully realized before! Fully half your prospects are interested in property coverage. Perhaps you had better spend more time boning up on the details of providing property insurance and less time on life policies? In any event, you are now in a position to press a few more keys on the computer and have mailing labels printed for each group of prospects. You then stuff an appropriately slanted letter into each envelope to kick off your new, streamlined selling approach.

Now it is time for you to go on the road and call on all those prospects. But wait! Your territory covers three states. Are you just going to hop in your car and drive haphazardly from place to place? Not any more. You direct your computer to re-sort your data according to the contents of the "State" field. When grouped accordingly, you then sort each state subgroup against the "City" field. Presto! You can now have the computer print you a list of prospects ordered by city and state. You can immediately pinpoint those cities that have the most prospects. Now you can plan your trip for maximum selling effectiveness.

In a matter of a few hours your business lifestyle has changed from seeming chaos to ordered efficiency. What do you think of that?

Of course, this example was made very simple to illustrate the concept of how a data base management program can be put to work. In a realistic application, you might expand the number of fields in a record to, say, 2Ø or 3Ø, in order to keep track of a great many parameters. Thus, you might have a field for a client's phone number, fields for dates on which you actually made a call, fields amplifying a customer's special requirements, fields for recording policy numbers, renewal dates, etc. Thus, the power and efficiency of the system can be considerably amplified beyond that illustrated in this initial example. Put your imagination in gear. Do you have visions of personal power and control? Can you think of ways in which a system such as this could make your life more productive and enjoyable? I'll bet you do and that you can!

### Watch Those Expenses!

Do you know that you can use this same data base management program to help you keep track of a personal expenditures budget?

One way to approach such an application would be to format a record with the following fields: Item, Category, Tax Code, Budget Amount, Actual Amount.

At the beginning of each accounting period, let us say monthly for purposes of this discussion, you could enter projections or estimates. Some typical entries might appear as follows on the next page.

```
            RECORD NR. 1
      ITEM: MORTGAGE INTEREST
  CATEGORY: MORTGAGE
  TAX CODE: DEDUC
BUDGET AMT: 550
ACTUAL AMT:
            RECORD NR. 2
      ITEM: MORTGAGE PRINCIPAL
  CATEGORY: MORTGAGE
  TAX CODE: NON-DEDUC
BUDGET AMT: 22
ACTUAL AMT:
            RECORD NR. 3
      ITEM: ELECTRICITY
  CATEGORY: UTILITIES
  TAX CODE: NON-DEDUC
BUDGET AMT: 45
ACTUAL AMT:
            RECORD NR. 4
      ITEM: PHONE SERVICE
  CATEGORY: UTILITIES
  TAX CODE: NON-DEDUC
BUDGET AMT: 80
ACTUAL AMT:
```

You would continue making similar entries for whatever expenditures you anticipated making. When you had completed indicating your plans, you could have the computer "tally" the total budgeted expenditures for all the records under the field named "Budget Amt." Will you have enough money for the projected expenditures? If not, you might want to alter your spending plans on the spot.

You could save this initial projection on a diskette. Then, as you actually made expenditures or at the end of the month, you could recall the data base from the diskette. You would then enter the actual expense

amounts under the field named "Actual Amt." Now you could compare how your projections did versus actual charges. And, you could tally the "Actual Amt" field to obtain a total for the period too.

But that is not all. Suppose you wanted to look at your total utility expenses? All you need do is perform a sort on the contents of the "Category" field. Then, with all expenses for "utilities" grouped together, you could direct the program to tally over just the selected "utilities" expenditures. Spending too much there? Put a stop to it right now, *before* you go broke!

And how about at the end of the year when it comes time to pull together all those tax deductions? Well, you could perform a search operation on the contents of the "Tax Code" field and have the computer display all those beautiful tax deductions. Or, you could sort on that field and have the computer group all deductions together in the file. Then you could have the computer perform a mathematical tally of all the records in that group to obtain a total "tax-deduction" figure! Neat, eh?

Are you beginning to see the big picture? Let's try one more hypothetical illustration:

You are a small businessperson. Naturally, you have to give credit terms to your regular customers. But, it turns out that not everybody pays promptly. You need to keep track of who owes you money!

You format a file with fields to hold: the date of a transaction, invoice number, description of merchandise, amount due, customer's name, address, etc.

Each day you enter the information pertaining to sales made on credit. If accounts are paid, you simple remove the appropriate record from the file, since you no longer need to track the invoice as an account receivable.

Perhaps you want to remind customers of past-due invoices at the end of each month? You could arrange the file according to serial invoice numbers, for example, by performing an appropriate sort operation. You could then have the computer list the invoice number, amount due, and name and address of each account that is in arrears. Why, you could pop the slips of paper right off the printer into a glassine-window envelope and have a friendly reminder in the mail in a matter of seconds! Ah, the wonders of computer technology and the power of a data base management program.

Are you ready to put a data base manager to work for your own sake? Then just turn to the next chapter to get into the nitty-gritty details of operating your own Apple data base program.

# 3 OPERATING THE DATA BASE PROGRAM

In this chapter I will try to give you a feel for how you communicate with and give directives to the data base management program described in this book and presented at the end of this chapter. Later portions of this manual will provide plenty of practical examples of its operation.

As you read this chapter, you may want to try various operations yourself. If that is the case, then load the program into your Apple II, Apple II Plus, or Apple IIe computer and start it by giving the computer the RUN command.

The minimum equipment you need to successfully implement this program is an Apple II computer equipped with at least 48K of memory and one disk drive. The system must be capable of executing Applesoft BASIC. You will be able to build larger data files if the Applesoft interpreter is in read-only memory (ROM) rather than random-access memory (RAM). Of course, you will also need a system display device. If you want hard copy of your data base, then you will have to have a printer as part of your system. The program expects to see the printer interface in slot number 1 of the Apple system.

## It Is Interactive!

This data base management program is designed to respond to the user in what some people describe as a "conversational" manner. That is, it acts to prompt and guide the user. It does this by providing menus from which one makes choices, or by asking questions when it expects input(s) from the operator. This continuous *interactive* mode of operation provides a user-friendly environment. Such operation is beneficial to novice users, since it removes much of the confusion that can occur when initially using a computer program. Yet, the system retains a high degree of efficiency for experienced users who may not need much assistance.

In actual practice, operation of the program consists of really nothing more than making choices presented by the computer or responding to requests for inputs. It is thus not at all difficult to physically operate the program.

Of course, applying the program effectively requires learning some fundamental principles about data base management. You will gain much of the understanding needed through actual practical application examples provided later in this book. For now, let's get started with an overview of the key capabilities of the program.

### The Primary Menu

When you first start the program, a menu will appear on the screen with the choices:

```
1. USE APPLE DATA BASE
2. DEFINE RECORD FORMAT
3. READ FILE FROM DISK
4. WRITE FILE TO DISK
5. READ DISKETTE CATALOG
6. EXIT TO DISK EXECUTIVE


9. ERASE FILE FROM MEMORY
```

This menu will be referred to as the *Primary Menu*, since it allows the user to select the broadest types of operations.

To make a choice, you merely press the digit key corresponding to the number shown in the menu. Pressing the number 5, for example, results in the computer presenting a catalog of the diskette currently in the system disk drive. Note that, when selecting a menu option, it is only necessary to press the single key corresponding to the choice desired. In fact, the pressing of any key other than the digits 1 to 6 and the number 9 is ignored by the program when the Primary Menu is being displayed.

### Creating a Data File

Naturally, when using a computer, you cannot simply sit down and blindly start typing in data. The first thing you must do when using this data base program is to define a file format. That is, tell the computer how many fields, what type of fields, the size (in terms of numbers of characters) of each field, and an appropriate reference name for each field in a record. You begin inputting this information by selecting option number 2 (Define Record Format) of the Primary Menu.

When option number 2 is selected, the program proceeds to ask the following series of questions, in the order given: a name for a field, the maximum number of characters that will be permitted in the field, and whether the field is alphanumeric or strictly numeric.

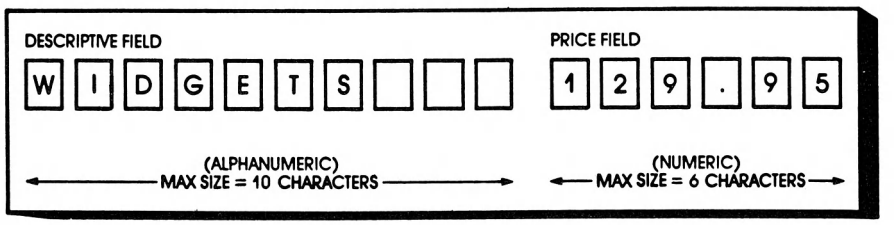


**Fig. 3-1. Records are formatted by user-defined fields. Each field is assigned a name, a maximum length, and is defined as alphanumeric or numeric.**

You input the appropriate information for each field. Then the program asks if you want to establish another field. If so, the same information is requested again. This procedure is followed until all the fields necessary for a particular application have been defined.

There are program limitations that must be adhered to when defining a file structure. For instance, field names may not exceed 1Ø characters. The maximum length of any field may not be defined as greater than 4Ø characters. Fields may only be designated as (A)lphanumeric or (N)umeric in type. A record may not contain more than 4Ø fields. And the maximum number of characters permitted in a record (obtained by summing the maximum number of characters permitted in all the fields in a record) is limited to 236.

Should you attempt to exceed the program limitations mentioned, the program will remind you of its capabilities and give you the opportunity to make amends.

Once you have completed defining all the fields that are to be provided for in a record, the program determines the maximum number of records that can be held in a file.

Remember, this is a "memory-resident" data base management program. That is, all of the information within a specific file actually resides within the RAM of the computer when a file is in use. Thus, the size of a file and the number of records it can handle are functions of the amount of memory in your system and the length of the defined records.

The maximum number of records that a file can hold is calculated by the program immediately after the last field has been defined. This value is displayed briefly to the user for reference. Should you discover that a particular file definition does not provide for enough records for a particular application, you might want to consider redefining the file structure in order to achieve your goals. Decreasing the number of fields or the number of characters permitted in fields will increase the number of records that may be stored in a file. Up to a point, that is. The maximum number of records that can be stored in any one file is limited by the program to 999.

(If you need more records than this in a single file, you are probably well advised to look into obtaining a disk-oriented data base management

program. That is one where each record is actually stored on a diskette and access to the diskette is made each time any information in the data base is required. Such disk-oriented data base management programs are generally quite expensive.)

### The Secondary or Operations Menu

Once you have defined the structure of a file, you may begin inputting information to build up a data base. At the conclusion of the file-defining process, the program will automatically go to the Operations Menu. (This will also sometimes be referred to as the Secondary Menu, since it is not as broad in scope as the Primary Menu.)

You can also get to the Operations Menu by selecting option number 1 (Use Apple Data Base) when the Primary Menu is on the screen. Of course, if you try to use this option before you have defined a file (or read a previously stored file in from a diskette), then your directive will be politely ignored by the program. After all, you can't use a data base if you haven't at the very least defined its format.

Here is how the Operations (Secondary) Menu appears on your screen:

```
1. APPEND RECORD(S)
2. INSERT RECORD(S)
3. CHANGE RECORD(S)
4. DELETE RECORD(S)
5. LIST   RECORD(S)
6. FIND   RECORD(S)
7. SORT   RECORD(S)
8. TALLY  RECORD(S)
9. ** NEXT  MENU **
```

Those are the data base *operations* that may be performed by this program. Option number 9 returns you to the Primary Menu. (It could direct you somewhere else if you are the adventurous type. Those with programming ability might want to modify the program to give it further capabilities. Well, there's a "hook" for you to use if you want to hang your own stuff onto the program!)

### The APPEND Operation

The first thing you have to do in order to start working with a computerized data base is to load the data into your system. The APPEND option of the Operations Menu is used to place the program in the mode to accept and *append* new data to the end of the current memory-resident data file.

On the top of p. 20 is an example of how the screen might appear in a typical application when the APPEND option was selected.

```
--- APPEND RECORD(S) ---

RECORD # 4   MAX CHARS = 20   ALPHA

FIELD # 1 NAMED: NAME
-----------------------------------------------
```

Note that the screen reminds the user of the type of operation being performed. In this example, it indicates that the system is ready to accept data for the fourth record in a file. The current field is limited to a maximum of 20 characters of information. The field type is alpha-numeric. Data is to be inputted for the first field in the record and that field is called the "Name" field. The operator (that's you, isn't it?) will then type in the information requested. That new data will appear beneath the dashed line that is used to separate the computer's prompting from the operator's input.

Simple, isn't it?

As soon as the information to be stored in one field has been entered (by typing it on the keyboard and pressing the ENTER key), the program automatically updates the display to prompt for the data for the next field. This process continues until all of the fields for one record have been accepted.

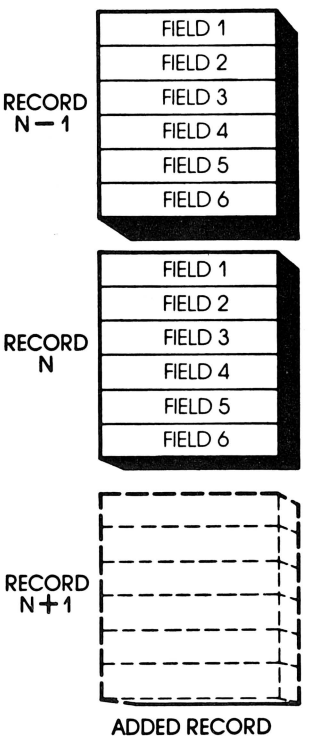


**Fig. 3-2. The APPEND operation adds a record at the *end* of a file.**

When the information for one record has been processed, the user is asked if another record is to be processed. If so, then the display is updated to prompt for the first field in the next consecutive record. If not, then the APPEND operation is terminated by the program returning to the Operations Menu.

You can continue inputting new records into a file until it contains the maximum number allowed. (Remember, this will be a function of the amount of memory available in the system as well as the length of the defined records.) Should you fill the file while in the APPEND mode, the program will automatically terminate this mode and return to the Operations Menu. Should you attempt to select the APPEND mode when the file is already filled, then you will be notified that the "file is full" and the program will not attempt to perform the operation.

Of course, during the inputting of information, the program performs certain checks on the validity of the information being entered. For example, if a field is limited to 2Ø characters, the program will not allow the operator to input a string that exceeds that length. Similarly, if a field has been declared to be numeric in type, then it had better only contain a *number*. Otherwise, the program will advise the user that the data cannot be accepted and provide the opportunity to correct the situation.

### Numeric Fields Only Accept Significant Digits

If you want to enter the numeric value, say, "2" in a field that has been defined as numeric in type, then you only enter the digit 2—you must not attempt to enter 2.Ø or 2. or Ø2.—since these three latter forms of the value all contain nonsignificant digits or characters. Furthermore, this program always assumes that a number is positive in value unless it is preceded by a minus (−) sign. So, do not attempt to use the plus (+) sign in a numeric field.

These restrictions in the definitions of numeric values as they apply to numeric type fields are made necessary by the manner in which the program checks for operator errors. There are tradeoffs to be made in the design of any program. In this case, the protection of novice users was given weight over the flexibility some more experienced users might enjoy. But then it is likely that the "pros" will have the wherewithal to go right in and modify the package if they are not happy with this particular limitation!

### The INSERT Operation

The APPEND operation previously discussed always adds or "appends" records to the *end of a file.* If there are three records in a file, it adds a fourth. When there are four, it adds the fifth, and so on.

Sometimes it is useful to be able to effectively *insert* a record in a file. That is, move all of the records beyond a certain point in the file, thereby leaving a space so that a record can be placed in the file. The advantage to this procedure is that the file can remain "ordered." Thus, it is particularly useful if a file has been sorted by some category and you want to add a new entry at a particular point in the neatly arranged file.

When the INSERT option is selected, the program initially responds with a display having the following format:

```
--- INSERT RECORD(S) ---

LAST RECORD IN FILE IS: 3

INSERT BEFORE RECORD NUMBER?
```

Note that this option starts off by advising the user of the highest numbered record that currently exists in the file. This is done for the sake of reminding the user of the highest possible position at which a new record could be inserted.

Once a user specifies the record number at which a new record is to be inserted, the program switches to a display format similar to that used by the APPEND option. This format prompts the user for the data that is to be inserted. If you were inserting a new record at record position number 2 in a file, a typical INSERT display might appear as:

```
--- INSERT RECORD(S) ---

RECORD # 2   MAX CHARS = 20   ALPHA

FIELD # 1 NAMED: NAME
----------------------------------------
```

Note that, by giving the record *before* which the new record is to be inserted, the inserted record actually acquires that record number. Thus, stating that you want to insert *before* record number 2 means that the inserted record becomes record number 2 in the file.

When all the fields in the record being inserted have been processed, the program asks if the user wants to continue inserting records *at the same point in the file.* If so, then the program prompts for the data that is to go in the next inserted record.

It is important to realize here that when a number of records are inserted at one place in the file, that the inserted records will acquire consecutive numbers. All other records in the file are moved apart ("up" in terms of record number assignments) to make room for the inserted records. Thus, specifying the first insertion point as before record number 2 and consequently providing data for two records result in those records becoming record numbers 2 and 3 in the file. The original record number 2 would be moved "up" to become record number 4 in the

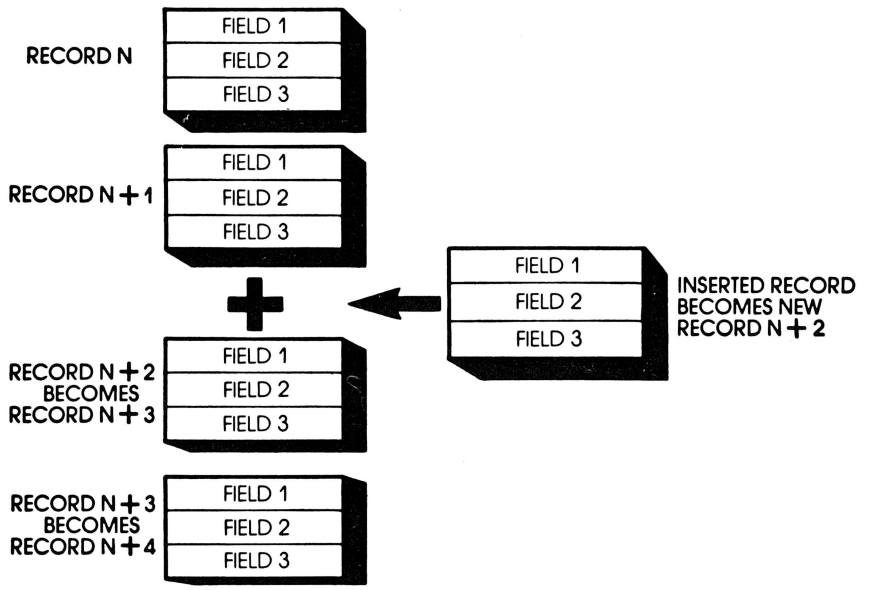


**Fig. 3-3. The INSERT operation inserts a record *within* a file.**

file. All other original record numbers higher in number would also be moved "up" appropriately.

## The CHANGE Operation

After a file of data has been created, it frequently needs to be updated. For example, if you are maintaining a mailing list, it is a sure bet that you will constantly be having to revise addresses due to people moving.

You use the CHANGE option to make alterations to records already in a file.

Now, the CHANGE capability of this data base management program will take a little longer to explain than some of the other operations. This is because it has some special features that, while taking a little longer to initially explain, result in faster and smoother operation when you need to make a lot of changes in a file.

Experience has shown that in typical applications only a few fields in each record need to be changed. However, it is often necessary to change a number of records within a file.

Taking a mailing list application for further illustration, suppose you receive notices of changes in addresses on a daily basis. Once a week or month or whatever period of time is convenient, you sit down to update your computerized data base. You know, as you prepare to make a batch of address updates, that only the street address, city, state, and zip code fields will be altered. It will not be necessary to alter a person's name or

classification or any of the other data you might be keeping in each record.

The most efficient way to tackle this job would be to just modify those fields that need alteration in the appropriate records.

That is exactly how you can proceed using this data base management program! (Some data base management programs do not have this kind of flexibility.)

When you first select the CHANGE option, the display will be formatted along the following lines:

```
--- CHANGE RECORD(S) ---

INPUT FIELD(S) TO BE CHANGED.
USE FIELD NUMBERS (NOT NAMES).
END YOUR LIST BY INPUTTING THE # Ø.
```

If, for illustrative purposes, field numbers 2, 3, 4, and 5 contained the street address, city, state, and zip code, respectively, within the records of a file, you would then input those numbers at this point. (After each number is typed you would press the ENTER key to input it to the program.) Note that you terminate your list of fields to be changed by entering the number Ø.

What you are doing is setting up the data base management program to deal *only* with those fields that need changing. It will thus skip over the fields that are to remain unaltered.

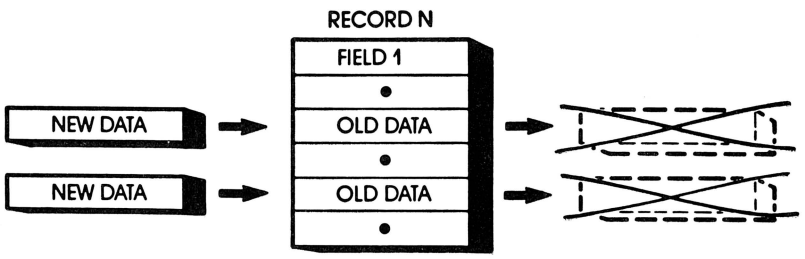


**Fig. 3-4. The CHANGE operation permits the contents of individual fields to be modified.**

Once you have indicated which fields within records are to be modified, the program proceeds to ask for the number of the first record in the file that is to be updated:

```
--- CHANGE RECORD(S) ---

LAST RECORD IN THE FILE IS: 3

BEGIN CHANGES WITH RECORD NUMBER?
```

Note that this portion of the program also reminds you of the highest numbered record currently in the file. After all, you don't want to start trying to make changes to a nonexistent record.

When you have indicated the number of the record in the file that you want to start changing, the display responds with the familiar prompting format:

```
--- CHANGE RECORD(S) ---

RECORD # 2   MAX CHARS = 40   ALPHA

FIELD # 2 NAMED: ADDRESS
-----------------------------------
```

As you input the new data for each field, the display will prompt for the next field that is to be changed. When all of the designated fields within a record have been processed, the program will ask:

```
CHANGE NEXT RECORD?
```

If so, the program automatically brings up the next record in the file and prompts for changes to the same set of fields. Thus, you can rapidly proceed to make alterations to a block of records within a file.

If you do not desire to alter the next record in the file, then the program terminates the CHANGE mode and returns to the Operations Menu.

### The DELETE Operation

When you want to remove one or more records from a file, you use the DELETE option. This directive is effectively opposite to the INSERT directive. When you remove a record using this command, the space originally occupied by that record in the file is physically recovered as all of the data above that point is moved "down." (The length of the file is thus shortened, making room for new data, if desired.)

When the DELETE option is selected, you are first reminded of the highest numbered record in the file. Then you specify the number of the record you wish to remove. You can give directives to have a single record eliminated or an entire group taken out. This is how the display would appear when deleting a single record from a file:

```
--- DELETE RECORD(S) ---

LAST RECORD IN FILE IS: 3

FIRST RECORD TO PROCESS?
2

LAST RECORD TO PROCESS?
2
```

Note that the "first" record and "last" record number are the same in this example, since only a single record is being removed.

If you wanted to remove an entire block of records, then you would indicate a range of record numbers. When indicating that a group of records is to be deleted, you must always proceed from lower numbered to higher numbered records. As a precaution against accidental erasure of records due to operator error, the program does not accept record number inputs that do not follow this order.

Furthermore, since the delete operation can be considered critical (it could "undo" a lot of work if invoked by accident!), you must confirm the record numbers being deleted before the operation actually occurs. Thus, after you have specified which record (or records) is (are) to be removed, the program asks for confirmation of your directives in the format shown here:

```
YOU HAVE REQUESTED THE DELETION
OF RECORD NUMBER(S) 2 THROUGH 2

IS THIS CORRECT (Y/N)?
```

If you respond (Y)es for "affirmative" here, then the deletion operation is performed. All higher numbered record(s) left in the file

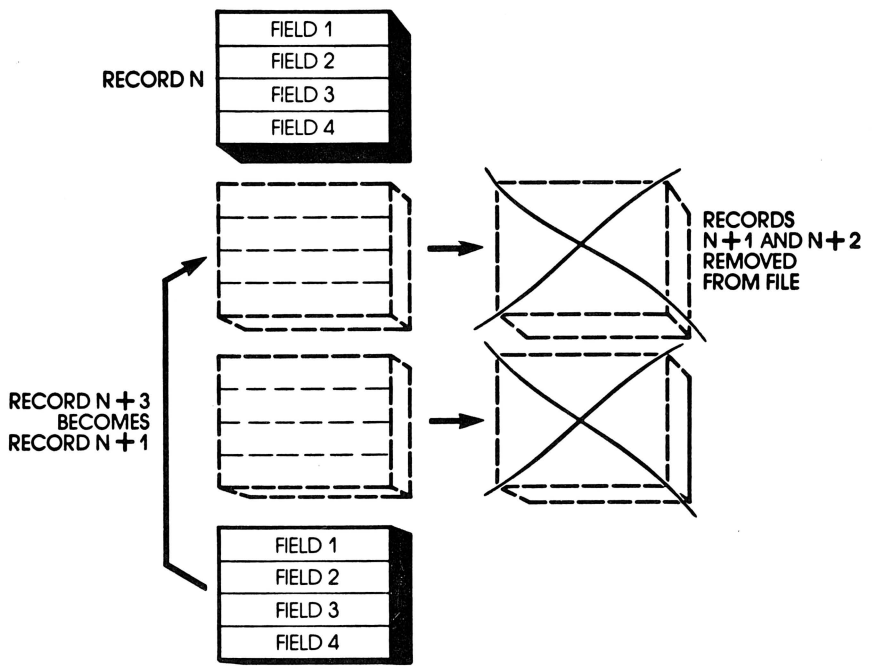


**Fig. 3-5. The DELETE operation removes entire records from a file.**

after the unwanted records have been removed are moved "down" to fill in the vacated area. Then the system displays how many records remain in the file. Thus, if you initially had three records in the file and you deleted a single record, you would see the message

```
LAST RECORD IN FILE IS: 2
```

appear on the screen.

If you reply to the confirmation query with anything other then (Y)es, the deletion operation is aborted. Control then returns to the Operations Menu, as it does when a deletion operation has been completed.

It is also worth noting that the DELETE option can serve a special purpose when formatting files. Suppose you wish to establish a whole group of files, all of which have exactly the same record format. As you already know, the first step you have to take is to define the file format. However, once you have done that for one file, you can save yourself a lot of work by using the DELETE option. When you have finished working with the first file—say after filling it up with data—you save it on a diskette (that process will be explained later). Then you select the DELETE option and *delete all of the records in the current, memory-resident file*! Presto! You are instantly ready to start loading data into another, identically formatted file. (You will see later that you can save this "new" file on diskette under a different name, etc.) There is no need to go through the file-defining process again, since you already have precisely the file format you need residing in memory. Remember this little tip. It can save you some work in future applications!

### The LIST Operation

Up to this point, I have been describing commands that let you construct a file and put data into it, alter the data or remove it. But, if that were all you could do, you soon would get pretty bored with your data base management program. The program would be just about useless, too. After all, why stash all that information in a computer file, if the machine won't save you some work!

It will. The LIST option is the first directive to be described that can really show you some practical results. Basically, what it does is enable you to list all or parts of a data file. It also gives you control over which fields in a record are to be outputted when performing a list operation. As you will soon see, these capabilities can provide some very practical results.

When first selected, you are informed of the highest numbered record in the file and asked which records you want to see. The initial display has the format illustrated on the top of the next page.

```
--- LIST   RECORD(S) ---

LAST RECORD IN FILE IS: 3

FIRST RECORD TO PROCESS?
2

LAST RECORD TO PROCESS?
2
```

In the example display, only one record is being called for, so the "first" and "last" record numbers to be processed are identical. If a group of records was desired, then the numbers would be different. The "last" record number must not be less than the "first" or the program assumes an error has been made by the user, whereupon the program asks the operator to try again.

As will be illustrated later in this book when practical examples of this program's usages are demonstrated, the ability to output a given range of records is highly useful. There are many times when dealing with a file of data that only a portion of the file's contents are desired. The operation of this LIST option provides the flexibility required in this regard.

Further flexibility in the outputting of information is garnered from the next step in the listing process. At this point, the program will query:

```
WANT TO SUPPRESS ANY FIELDS (Y/N)?
```

Responding affirmatively here (by typing the letter Y for *yes*) results in the program displaying the prompt:

```
INPUT FIELD(S) TO BE SUPPRESSED.
USE FIELD NUMBERS (NOT NAMES).
END YOUR LIST BY INPUTTING THE # Ø.
```

Now you can specify the field numbers whose contents you do *not* want to have outputted during the list operation. You will undoubtedly find yourself using this feature quite often. For instance, when you just want to extract specific information. (Suppose you have a file wherein each record holds a person's name, street address, city, state, zip code, telephone number, and perhaps several other fields containing amplifying data. You decide it would be helpful if you had a convenient list of all the people in this file and their phone numbers, *without any of the other data.* Fine, you just suppress the outputting of all the fields except for the "name" and "telephone number" fields. There you are!)

After you specify which fields are to be suppressed (if any), the program asks the following (see top of next page):

```
SUPPRESS RECORD NUMBERS (Y/N)?
```

This query provides the option of having the actual, physical record number displayed along with the information being outputted for each record. (This "physical" record number is based on the actual position within the file that the record occupies at the time the listing takes place.)

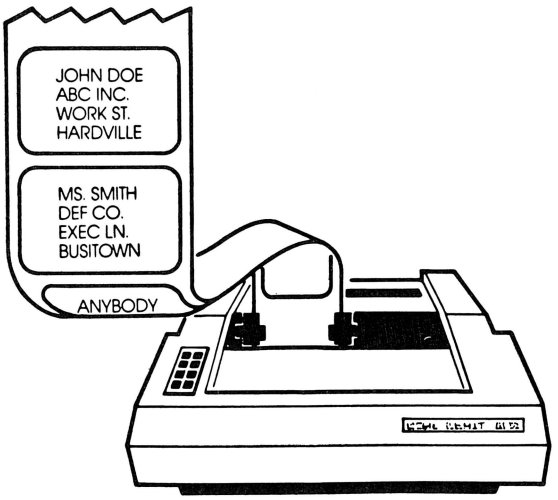


**Fig. 3-6. The LIST option can be used to print labels by suppressing fields that are not needed in the listing.**

The ability to list record numbers is important when one plans to edit or modify files or to arrange data within files into groups. If you have a mailing list file containing many names and addresses, you need to know the actual record numbers so that you can quickly and easily update or delete specific records.

However, at other times you may not want to have the record numbers listed. For instance, if you just want a list of names and telephone numbers, then you don't *need* to know the record numbers. In fact, they would just clutter up your listing, so you can elect not to have the record numbers displayed.

Next you have the option of having information listed on your video monitor or your system printer. You select the desired output device by responding to the query:

```
OUTPUT TO SCREEN OR PRINTER (S/P)?
```

appropriately.

If you elect to have the information listed on the screen, you will have an additional option as the information is displayed. Due to the fact that information is presented to the screen rapidly, the program stops outputting information after the data in each record is displayed. You are given as much time as you like to review the information. When you want to go on to the next record, you press any key *except* the RETURN key. Pressing the RETURN key when listing to the screen aborts the mode. This allows you an easy "out" if you tire of viewing records on the screen.

When information is listed on a printer, the entire range of records specified is outputted without interruption.

Regardless of which device the information is outputted to, the listing will be preceded by a "header." It shows the format of the listing by giving the field number, field name, its size (number of characters permitted in the field) and type (whether (A)lphanumeric or (N)umeric) for all the fields that will be outputted. This information is presented as a reminder to the user of just what data are contained in each field. It is also useful information to have when performing editing of the data base.

(Note: If you ever "forget" the format you are using in a particular file, just use the LIST option to output *any* record. The "header" obtained when this is done will refresh your memory in short order. Just be sure you don't suppress any fields when using the LIST option for this purpose!)

Here is what the header and a single record might look like when using the LIST option to output data to a printer:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  NAME                          20 A
 2:  STREET                        30 A
 3:  CITY                          12 A
 4:  STATE                          2 A
 5:  ZIP CODE                       5 A
 6:  TELEPHONE                     14 A
 7:  CATEGORY                       8 A
 8:  CODE                           4 A


JOHN DOE
123 PLEASANT STREET
ANYTOWN
NJ
07777
(201) 999-1234
PROSPECT
ABCD
```

OK, that is the basic LIST operation. Now let us move on to learn about a sophisticated variation . . . .

## The FIND Operation

Often, when dealing with a sizable data base, it is desirable to be able to "search" for items of specific interest. One way to conduct such a search is to list all or part of a file, then use the old "eyeball" technique. But really now, why did you get a computer? Why, to have it do that kind of looking *for* you!

The FIND option allows you to search any field in the data base for information of interest. This option starts off just like the previously described LIST option. When invoked, the program first informs you of the highest numbered record in the file. It then asks for the range of record numbers to be processed. That is, over what group of records is the search operation to be conducted. You can specify the examination of a single record or any continuous group. When specifying the latter, you must give the lowest numbered record, then the highest numbered record to be scanned.

Then, just as in the LIST option, you are given the opportunity to suppress the output of selected fields, if desired. Fields to be suppressed

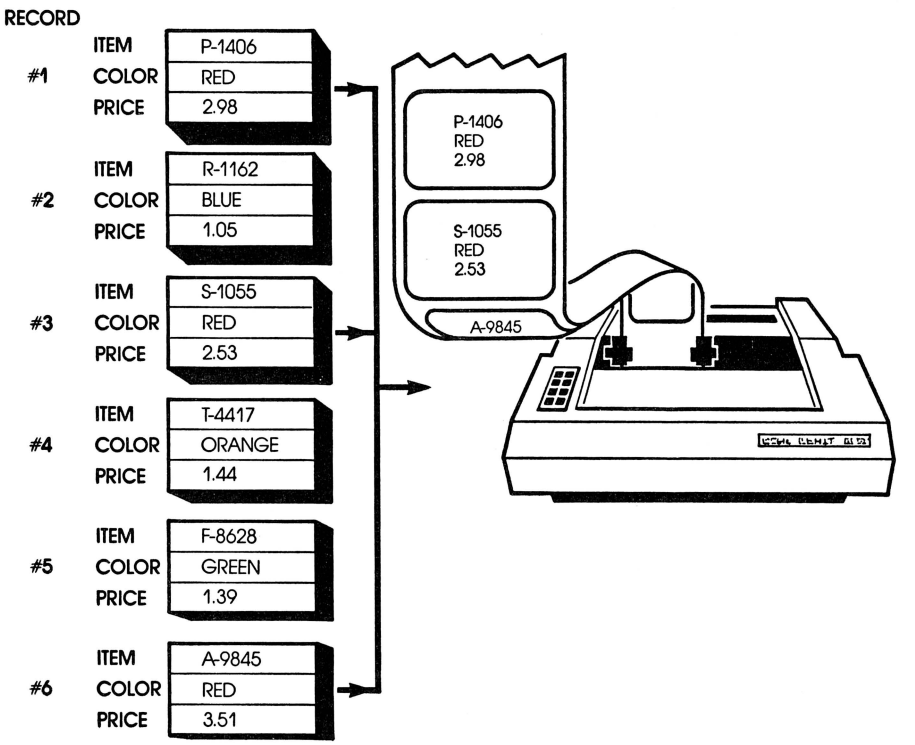


**Fig. 3-7. The FIND option can locate items of particular interest. For example, in the inventory file above, all items of the file having the color red have been located by the FIND option.**

are indicated by their position (number) within the record. The field number Ø is used to terminate the specification process.

And, once again, you are given the chance to suppress the displaying of record numbers when data is outputted, if desired.

Once all of these options have been decided, exactly as may be done for the LIST operation, the program proceeds to ascertain what is to be sought in the data base. It starts this process by asking:

```
SEARCH FIELD NAMED?
```

You input the *user-assigned name* given to the field that is to be searched within each record. For instance, in a mailing list application, fields might be named Name, Address, City, State, and so forth. If you were going to look for records containing addresses in a particular state, then you would specify the field named State at this point.

Suppose, however, that you are forgetful or careless. You cannot remember the exact name of a particular field or you misspell a field name. As the saying goes, "no problem." If the program receives a name it does not recognize, it will tell you what the valid field names are for the current file. Here is how the display might appear if you inputted a field name that did not exist in a file:

```
NO FIELD NAMED: STATT

VALID FIELD NAMES ARE:

NAME
ADDRESS
CITY
STATE
ZIP CODE
TELEPHONE
CATEGORY
CODE
```

You would then be given another opportunity to specify a valid field name.

Once an acceptable field name has been given, the program asks:

```
LOOK FOR A MATCH WITH?
```

You can then enter any string that you want the program to look for *anywhere* in the specified field.

Suppose, for instance, that you wanted to locate all the entries in a mailing list that were in the state of California. Assuming that you were abbreviating the state using the official post office abbreviations, you could request that the program look for a match with "CA" in the field named State.

Once this specification had been made, the program would proceed to examine the specified field in each record in the file (over the range of record numbers requested). It would look for a match *anywhere in the field* with the character string (CA in this hypothetical example) stipulated. If a match was found, then the contents of the entire record (less any suppressed fields) would be displayed.

As in the case of the LIST operation, if the data is being outputted to a video display, then the program pauses after each record has been displayed. It remains halted until the user presses any key except RETURN. If the RETURN key is depressed, the FIND operation is terminated. Pressing any other key causes the program to continue its search.

On the other hand, when a printer is being used, the program will proceed to search the entire range of records requested without interruption. Data is outputted to the printer whenever an appropriate match is found.

The "header" at the start of output during a FIND operation is slightly different than during the LIST procedure to provide some differentiation. Here is how the output might appear in a typical application of the FIND operation. (This example assumes a record format identical to that used for illustrating the LIST operation. However, it is further hypothesized that the user elected to suppress the printing of fields 2, 3, 5, 6, 7, and 8.)

```
SEARCH OF FIELD NAMED: STATE
IN RECORDS 1 THROUGH 20
FOR THE STRING: CA

FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1: NAME                                20 A
 4: STATE                                2 A

( 8 )
HENRY SOMEBODY
CA

( 14 )
JANE SOMEONE
CA
```

The FIND option is a powerful command. Some ramifications and examples of its use in specific applications will be highlighted later in this book.

### The SORT Operation

Perhaps the most powerful capability of this data base management program, in the opinion of many users, is its ability to order the contents of a file. The user can give directives to have all or part of the file

arranged in alphabetical order (or numerical order, when dealing with numeric fields). The ordering process is based on the contents of a specified field.

The ability to specify the field on which ordering is to take place *as well as the ability to select what group of records is to be examined* within a file combine to form a means of providing what, in computerese, is sometimes referred to as multi-key sorting. More on this aspect later.

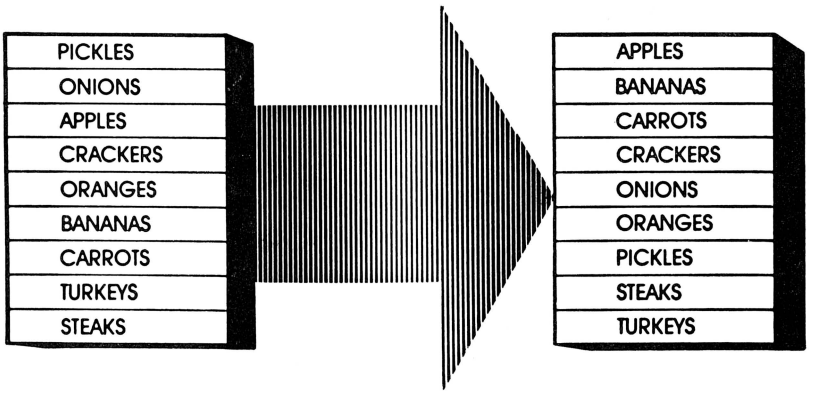


**Fig. 3-8. The SORT option enables records to be arranged according to the alphabetical (or numeric) contents of a specific field.**

This option also starts off similarly to the LIST and FIND operations. The user is immediately informed of the highest numbered record in the file, then asked for the range of records to be processed. The range specified must cover at least two records. Specify the lowest numbered record, then the highest.

Since the SORT operation only arranges the file within the memory of the computer, it is not necessary to specify any output options. You use the LIST option after a sort in order to see the ordered file or any section of it.

Once the range of records to be sorted has been indicated, the program queries:

```
SORT ON FIELD NAMED?
```

Just as when using the FIND option, you enter the name assigned to the field of interest. (That is, the name created by the user when the file format was established.) If the program does not recognize the name inputted at this point, then it lists the valid field names and allows you to try again.

Once a valid field name has been accepted, the program proceeds to sort the indicated records according to the contents of the specified field.

During the sort operation, a message having the format illustrated here will be displayed:

```
SORTING RECORD NUMBERS 1 THROUGH 3Ø
```

This alerts the operator to the fact that the computer is busy. A sorting operation can take from just a few seconds to a number of minutes. The actual time required is a function of the number of records being sorted, as well as the length of the field on which the sort is being conducted and how ordered the records are when the sort is initiated.

If the field upon which the sort is based is alphanumeric, then the records are ordered according to a character-by-character comparison of the entire field. The character ranking is based on the ASCII character code used internally by the computer. Records will be arranged in ascending character-code order.

If the field upon which the sort is based is numeric, then the records are ordered according to the signed numerical value of the contents of the field. Numeric sorts may take slightly longer, on average, than alphanumeric sorts.

When the sorting procedure has been accomplished, the program returns to the Secondary or Operations Menu. Use the LIST option if you wish to examine the file after a SORT operation.

## The TALLY Operation

The final "operation" provided in this data base management program is referred to as the TALLY option. It enables a user to take a tally (perform a summing operation) on a designated field over a range of

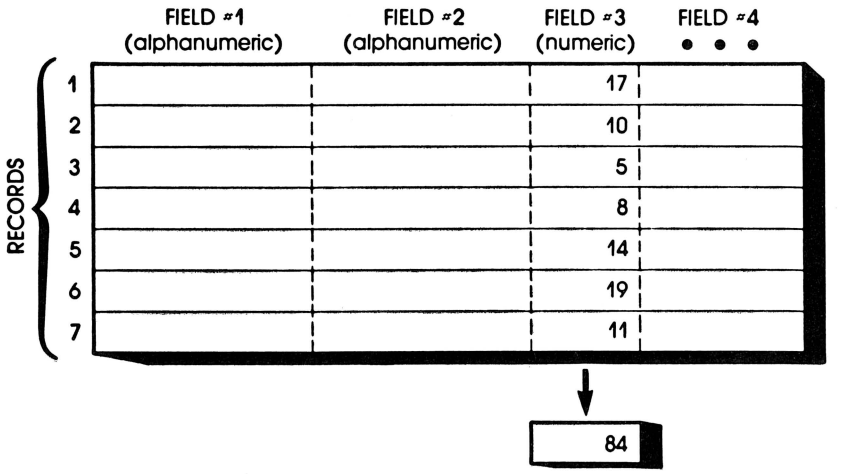


**Fig. 3-9. The TALLY option sums the contents of a specified numeric field in a group of records.**

records. *Only numeric fields may be tallied.* If a user attempts to perform a tallying operation on an alphanumeric field, the directive will be declined by the program.

Here is how the screen would appear at the beginning of a typical TALLY application:

```
        --- TALLY  RECORD(S) ---

LAST RECORD IN FILE IS: 30

FIRST RECORD TO PROCESS?
1

LAST RECORD TO PROCESS?
5
```

In this example, the user has specified that record numbers 1 through 5 are to be included in the tallying operation.

Next the user must respond to the prompt:

```
TALLY ON FIELD NAMED?
```

Note that the user-assigned name of a *numeric* field must be supplied.

Failure to give a valid field name results in the program reviewing the field names used in the current file. The user is then given another opportunity to input a valid field name.

Failure to specify a field name corresponding to a numeric (type) field, results in an appropriate reminder message. Since it is possible for a user to attempt to tally in a file with *no* numeric fields in it, a failure in this regard causes the program to default back to the Operations Menu. The user can then review the file format (using the LIST option, if necessary) and take appropriate steps to reinitiate the TALLY operation.

When a valid name for a numeric field has been verified, the program proceeds to sum the values in that field over the range of records requested. When the sum has been obtained, the result is displayed in the following format:

```
TALLY FOR FIELD NAMED NUMBERS
IN RECORD(S) 1 THROUGH 5 IS:

226179

PRESS ANY KEY TO CONTINUE....
```

The user can view the result on the screen as long as desired. When the user has finished evaluating the information, a press of any key concludes the TALLY operation. The program then returns to the Operations Menu to await a new directive.

### Saving a File on a Diskette

Whenever you want, you can save a copy of any formatted data file on a diskette. It is a good idea to get into the habit of doing this regularly.

For instance, suppose you were constructing a large data base. One with, say, several hundred records. It would be a good idea to stop after appending every 50 or so records, save the file on a diskette, then continue adding to the file. This type of procedure provides a measure of security against losing your data due to a power failure or other mishap.

You may also want to save copies of files arranged (sorted) by several different keys (field contents).

And, you might be interested in keeping copies of files in chronological order. You can do this by making a new copy of a file at the end of every week or other suitable period of time.

It is easy to make a copy of a memory-resident data file at any time using this program.

If the program is displaying the Operations Menu, you select option number 9 (Next Menu) to change the system to the Primary Menu.

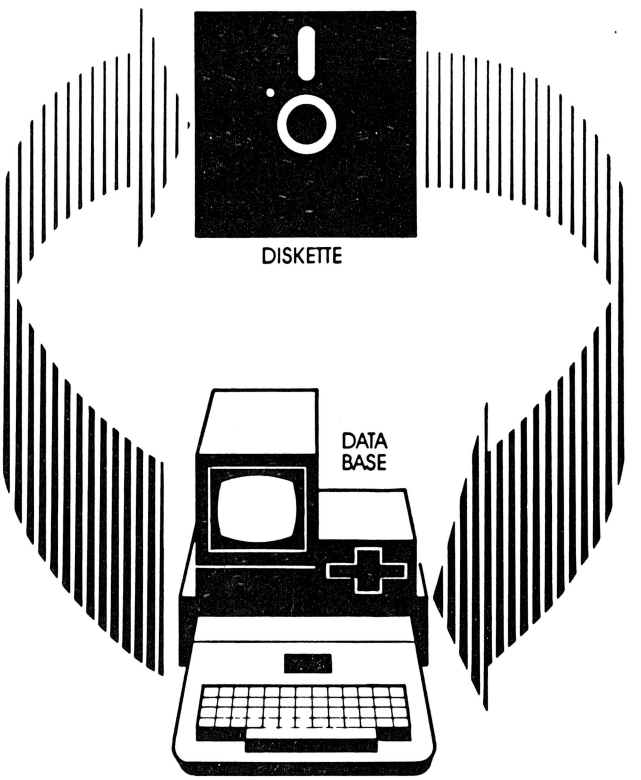


**Fig. 3-10. Data files may be transferred from the computer's memory to a diskette for long-term storage. They can be restored to memory from the diskette whenever desired.**

Once you have the Primary Menu on the display, just select option number 4: WRITE FILE TO DISK.

The program will respond with the following display:

```
--- SAVE FILE TO DISK ---

SPECIFY FILE NAME (MAX 30 CHARACTERS):
```

Type in the name you want to assign to the file when it is residing on the diskette. Make sure you have a diskette in your drive on which to save the file. Then, press the RETURN key. That is all there is to it. The program will store the current memory-resident data base file onto the diskette under the name you have assigned.

At the conclusion of the disk-write operation, the program returns to the Primary Menu.

If you should forget to place a diskette in your disk drive, if the diskette is full or write protected, or a system malfunction should occur during a disk input/output (I/O) operation, then the operation will be halted. You will then see the following message displayed:

```
I/O ERROR. RECOMMEND CHECKING DISK.

PRESS ANY KEY TO RETURN TO MENU....
```

In such a case, ascertain the reason for the failure, then press a key to return to the Primary Menu. If you were attempting to save a file to a diskette, it would still be in memory (provided the I/O failure wasn't caused by something catastrophic, such as a power failure). You can then simply correct the reason for the initial failure and try the procedure again.

You will want to remember to be careful when you assign names to files stored on a diskette. The data base program has been designed to overwrite any file on the diskette having the same name you assign when selecting the option. This was done so that you could keep updating the same file, if desired. If you want to have several copies of the same file on a diskette, then assign a different name each time you save it.

### Obtaining a Catalog of a Diskette

If you want to check the contents of a diskette, just select choice number 5: READ DISKETTE CATALOG of the Primary Menu. Doing so brings up the catalog of the diskette currently in the active drive. It is displayed in the standard disk-operating system (DOS) format such as that shown on the top of the next page.

```
DISK VOLUME Ø

 A  DATA BASE PROGRAM
 T  MAIL LIST

PRESS ANY KEY TO RETURN TO MENU....
```

When you are through examining the disk catalog, press any key to get back to the Primary Menu.

It is a good idea to call on this option whenever you are in doubt about the contents or status of a diskette. Using this *before* you attempt to, say, write to a diskette, can tell you whether a file name has been assigned previously. It can also be used to estimate how much room you have left on a diskette, etc. Remember, when in doubt, take a look at the catalog and find out!

### Reading a Data File from a Diskette

Of course, one of the powerful features of a computerized data base management system is its ease of use. You can design a structure for a data base. Fill it up with data. Manipulate the information to extract what you need at the present time. Then save its current contents on a diskette. Anytime you want to pick up where you left off, just load the data back into memory from the diskette and continue your work!

You already know how to save a data file on a diskette. Restoring a previously saved file to memory is just as easy. You just select option number 3: READ FILE FROM DISK of the Primary Menu. Doing so brings up the prompt:

```
--- READ FILE FROM DISK ---

SPECIFY FILE NAME (MAX 3Ø CHARACTERS):
```

It does, that is, unless you happen to already have a data file residing in memory!

Reading in a data file when you already had a file in memory would overwrite that data. Thus, if, when you select this option, there is already a file in memory, the program will ask:

```
DO YOU WISH TO ERASE THIS FILE (Y/N)?
```

This serves as a reminder that you already have an active file. If you respond (N)o to this query, then the program returns to the Primary Menu so that you can take action, say, to save the current file on a diskette.

If, on the other hand, you are through working with a particular file in memory and want to read in another file in order to work with it, then

you would respond (Y)es to the above query. Doing so will result in the present memory-resident data file being erased and the following message appearing briefly on the screen:

```
FILE HAS BEEN ERASED....
```

The program will then return to the Primary Menu *and you must reinstate the original directive (to read a file) again!* (Remember, this sequence has been designed as a safeguard to prevent a user from inadvertently overwriting a file that was present in memory.)

Assuming that no file was in memory when option 3 was selected, the user would simply type in the name of the file desired when the prompt

```
--- READ FILE FROM DISK ---

SPECIFY FILE NAME (MAX 30 CHARACTERS):
```

came up on the screen. After making sure the proper diskette was in the active drive, the RETURN key would be pressed to initiate the transfer process. As soon as the new data base has been successfully loaded into memory, the program returns to the Primary Menu.

Should any kind of I/O malfunction occur, the "I/O ERROR" message previously described would appear. The read operation would be terminated. Pressing any key would return the program to the Primary Menu. After action had been taken to clear up the I/O problem, the user would simply select the disk-read option to try the procedure again.

## Erasing a Data Base File from Memory

After working with a data base in memory, you may decide that you want to construct a new data base. If this data base is to have a new format, then you will need to select option number 2 of the Primary Menu. Before you can do this, however, you must erase the current file from memory. This is accomplished by selecting option 9: ERASE FILE FROM MEMORY of the Primary Menu.

As a protective measure to prevent accidental erasures, you must authenticate (verify) this option before the file will actually be deleted from memory. You do this by responding affirmatively to the following query:

```
DO YOU WISH TO ERASE THIS FILE (Y/N)?
```

If you do not respond with (Y)es here, then the program assumes a menu selection error was made. The menu is redisplayed.

If you do respond with (Y)es, then the current file is erased from memory and the message

```
FILE HAS BEEN ERASED....
```

is briefly displayed before the program returns to the Primary Menu.

### Leaving the Data Base Management Program

When you are through working with the program, you can return to the system's DOS executive by selecting option 6: EXIT TO DISK EXECUTIVE of the Primary Menu.

However, since puttering around with the DOS executive can wipe out any data base file currently in memory, this option, too, must be verified before it is actually executed. Thus, it is necessary to respond appropriately to the verification message:

```
RETURNING TO DISK EXECUTIVE DESTROYS
ANY DATA CURRENTLY IN MEMORY.

IS THAT OK (Y/N)?
```

If you do not confirm the option, the Primary Menu is redisplayed. Confirming it causes the data base program to be terminated. You should then see the DOS prompt appear on the display.

### Deleting Data Base Files from a Diskette

You remove data base files using your DOS executive. Use the DELETE command followed by the name of the file that you wish to have purged from the diskette. You can also COPY or RENAME your data base files using the DOS executive. Remember, these data base files are simply configured as "T" (text) files as far as the DOS is concerned.

### Try It!

That is the best way to become thoroughly familiar with the program. To get more ideas of how to apply your newfound computing power to practical applications, study and adapt the examples provided later in this book to serve your own personal needs.

The *Data Base Source Listing* follows on pp. 42-47.

## APPLE DATA BASE MANAGER: SOURCE LISTING

```
1  GOTO 40000
2  REM     (C) COPYRIGHT 1982
3  REM     SCELBI PUBLICATIONS
1000  IF AA$ = "" THEN 1030
1010  GOSUB 59980: PRINT "FILE ALREADY DEFINED....": GOSUB 59990: PRINT "E
     RASE CURRENT FILE BEFORE REDEFINING.":I = 2000: GOSUB 59970
1020  GOTO 40000
1030  CLEAR : GOSUB 1900
1100  GOSUB 59980
1110  PRINT "NAME FOR FIELD # ";X + 1;" (MAX 10 CHARS)?": PRINT
1120 LC = 10: GOSUB 50010
1130 A$(X) = W$
1200  GOSUB 59990
1210  PRINT "MAXIMUM # OF CHARACTERS IN FIELD ";X + 1;"?": PRINT "(LIMIT I
     S 40 CHARACTERS TO A FIELD)": PRINT
1220 LC = 2:LD = 1: GOSUB 51010
1230  IF W < 1 OR W > 40 THEN E$ = "OUTSIDE VALID RANGE. RE-ENTER.": GOSUB
     59800: GOSUB 59850: GOTO 1220
1240  IF B + W > 236 THEN E$ = "RECORD OVERFLOW. REDUCE FIELD LENGTH.": GOS
     59800: GOSUB 59850: GOTO 1220
1250 A(X) = W: GOSUB 59990
1260  PRINT "IS FIELD ";X + 1;" ALPHA OR NUMERIC (A/N)?": PRINT
1270 LC = 1: GOSUB 50010
1280  IF W$ = "A" THEN A(X + 40) = 0: GOTO 1310
1290  IF W$ <  > "N" THEN  GOSUB 59550: GOSUB 59800: GOSUB 59850: GOTO 127
     0
1300 A(X + 40) = 1
1310 B = B + A(X): IF B = 236 THEN 1400
1320  IF X > 38 THEN 1400
1330  GOSUB 59980
1340  PRINT "INPUT ANOTHER FIELD (Y/N)?": PRINT
1350 LC = 1: GOSUB 50010
1360  IF W$ = "Y" THEN X = X + 1: GOTO 1100
1370  IF W$ <  > "N" THEN  GOSUB 59550: GOSUB 59800: GOSUB 59850: GOTO 135
     0
1400  GOSUB 59980
1410  PRINT "YOU HAVE COMPLETED DEFINING THE FIELDS.": GOSUB 59990
1420 A(80) = X + 1:A(81) = 0
1430 C =  FRE (X) - 1000:A(83) = B:B =  INT (C / B): IF B > 999 THEN B = 9
     99
1440  PRINT "THERE IS ROOM FOR ";B;" RECORDS.":A(82) = B
1450 I = 1000: GOSUB 59970:AA$ = "D": GOTO 41000
1900  DIM A(83),A$(39),B(39),B$(999): RETURN
2000  GOSUB 2900: IF G = 1 THEN G = 0: GOTO 41000
2100 B = 1:X = 0:B$(A(81)) = ""
2200  GOSUB 59980:A$ =  STR$ ((A(81)) + 1): PRINT  TAB( 9);"--- APPEND REC
     ORD(S) ---": GOSUB 59990
2210  GOSUB 2800
2220 B$(A(81)) = B$(A(81)) + W$
2230  IF (X + 1) = A(80) THEN 2250
2240 B = B + A(X):X = X + 1: GOTO 2200
2250 A(81) = A(81) + 1
2260  GOSUB 59990: PRINT "CONTINUE WITH NEXT RECORD (Y/N)?": PRINT
2270 LC = 1: GOSUB 50010
2280  IF W$ = "Y" THEN 2000
2290  GOTO 41000
2800  PRINT "RECORD # ";A$;"   MAX CHARS = ";A(X);"  ";
2810  IF A(X + 40) = 0 THEN  PRINT " ALPHA": GOTO 2830
2820  PRINT " NUMERIC"
2830  PRINT : PRINT : PRINT "FIELD # ";X + 1;" NAMED: ";A$(X)
2840  PRINT "----------------------------------------": GOSUB 59990
2850 LC = A(X): IF A(X + 40) <  > 0 THEN LD = 0: GOSUB 51010: GOTO 2870
2860  GOSUB 50010
2870  IF  LEN (W$) < A(X) THEN  FOR I = 1 TO A(X) -  LEN (W$):W$ = W$ + "
     ": NEXT I
```

```
2880  RETURN
2900 G = 0: IF A(81) >  = A(82) THEN G = 1: GOSUB 59980: PRINT "FILE IS FU
     LL!!":I = 1000: GOSUB 59970
2910  RETURN
3000  GOSUB 2900: IF G = 1 THEN G = 0: GOTO 41000
3010  GOSUB 5800: IF G = 1 THEN G = 0: GOSUB 59990: PRINT "USE THE APPEND
     ROUTINE....":I = 1000: GOSUB 59970: GOTO 41000
3100  GOSUB 3900: GOSUB 5900: GOSUB 59990
3110  PRINT "INSERT BEFORE RECORD NUMBER? ": PRINT
3120 LC = 3:LD = 1: GOSUB 51010: IF W = 0 THEN 41000
3130  IF W < 1 OR W > A(81) THEN  GOSUB 59560: GOSUB 59800: GOSUB 59850: GOTO
     3120
3140  FOR I = A(81) TO W - 1 STEP  - 1:B$(I + 1) = B$(I): NEXT I
3150 B = 1:X = 0:A$ =  STR$ (W):B$(W - 1) = ""
3160  GOSUB 3900: GOSUB 59990: GOSUB 2800
3170 B$(W - 1) = B$(W - 1) + W$: IF (X + 1) = A(80) THEN 3200
3180 B = B + A(X):X = X + 1: GOTO 3160
3200 A(81) = A(81) + 1: GOSUB 2900: IF G = 1 THEN G = 0: GOTO 41000
3210  GOSUB 59990: PRINT "CONTINUE INSERTING RECORDS (Y/N)?": PRINT
3220 LC = 1: GOSUB 50010: IF W$ <  > "Y" THEN 41000
3230 W = W + 1: GOTO 3140
3900  GOSUB 59980: PRINT  TAB( 9);"--- INSERT RECORD(S) ---": RETURN
4000  GOSUB 5800: IF G = 1 THEN G = 0: GOTO 41000
4010  GOSUB 4900: GOSUB 6900
4020  GOSUB 59990: PRINT "INPUT FIELD(S) TO BE CHANGED."
4030  GOSUB 6800
4040  FOR I = 0 TO 39: IF B(I) <  > 0 THEN G = 1
4050  NEXT I: IF G = 1 THEN G = 0: GOTO 4100
4060  GOTO 41000
4100  GOSUB 4900: GOSUB 5900: GOSUB 59990
4110  PRINT "BEGIN CHANGES WITH RECORD NUMBER?": PRINT
4120 LC = 3:LD = 1: GOSUB 51010:Z = W: IF Z = 0 THEN 41000
4130  IF Z < 1 OR Z > A(81) THEN  GOSUB 59560: GOSUB 59800: GOSUB 59850: GOTO
     4120
4140 B = 1:X = 0:A$ =  STR$ (Z)
4150  IF B(X) = 0 THEN 4200
4160  GOSUB 4900: GOSUB 59990: GOSUB 2800
4170 B$(Z - 1) = "(" + B$(Z - 1) + ")"
4180 B$(Z - 1) =  LEFT$ (B$(Z - 1),B) + W$ +  MID$ (B$(Z - 1),B + A(X) + 1
     )
4190 B$(Z - 1) =  MID$ (B$(Z - 1),2, LEN (B$(Z - 1)) - 2)
4200  IF (X + 1) = A(80) THEN 4220
4210 B = B + A(X):X = X + 1: GOTO 4150
4220  IF Z = A(81) THEN 41000
4230  GOSUB 59990: PRINT "CHANGE NEXT RECORD (Y/N)?": PRINT
4240 LC = 1: GOSUB 50010: IF W$ <  > "Y" THEN 41000
4250 Z = Z + 1: GOTO 4140
4900  GOSUB 59980: PRINT  TAB( 9);"--- CHANGE RECORD(S) ---": RETURN
5000  GOSUB 5800: IF G = 1 THEN G = 0: GOTO 41000
5100  GOSUB 59980: PRINT  TAB( 9);"--- DELETE RECORD(S) ---": GOSUB 5900
5110  GOSUB 5700: IF G = 1 THEN G = 0: GOTO 41000
5120  GOSUB 59980: PRINT "YOU HAVE REQUESTED THE DELETION": PRINT "OF RECO
     RD NUMBER(S) ";Y;" THROUGH ";Z
5130  GOSUB 59990: PRINT "IS THIS CORRECT (Y/N)?": PRINT
5140 LC = 1: GOSUB 50010: IF W$ <  > "Y" THEN 5100
5200 F = Z - Y + 1: IF Z < A(81) THEN  FOR I = Z TO A(81):B$(I - F) = B$(I
     ): NEXT I
5210  FOR I = 1 TO F:A(81) = A(81) - 1:B$(A(81)) = "": NEXT I: GOSUB 5900:
      GOTO 41000
5700 G = 0: GOSUB 59990: PRINT "FIRST RECORD TO PROCESS?": PRINT
5710  GOSUB 5790: IF W = 0 THEN G = 1: RETURN
5720  IF W >  = 1 AND W <  = A(81) THEN 5740
5730  GOSUB 59560: GOSUB 59800: GOSUB 59850: GOTO 5710
5740 Y = W: GOSUB 59990: PRINT "LAST RECORD TO PROCESS?": PRINT
5750  GOSUB 5790: IF W = 0 THEN G = 1: RETURN
5760  IF W < Y OR W > A(81) THEN  GOSUB 59560: GOSUB 59800: GOSUB 59850: GOTO
     5750
5770 Z = W: RETURN
5790 LC = 3:LD = 1: GOSUB 51010: RETURN
```

```
5800 G = 0: IF A(81) = 0 THEN G = 1: GOSUB 59980: PRINT "FILE IS EMPTY!!":
     I = 500: GOSUB 59970
5810  RETURN
5900  GOSUB 59990: PRINT "LAST RECORD IN FILE IS: ";A(81):I = 500: GOSUB 5
     9970: RETURN
6000  GOSUB 5800: IF G = 1 THEN G = 0: GOTO 41000
6100  GOSUB 59980: PRINT  TAB( 9);"--- LIST   RECORD(S) ---": GOSUB 5900
6110  GOSUB 5700: IF G = 1 THEN G = 0: GOTO 41000
6200  GOSUB 6500
6210  GOSUB 59980: IF C = 1 THEN  PRINT  CHR$ (13) +  CHR$ (4) + "PR#1": GOSUB
     59990
6220  GOSUB 6600
6230  FOR I = Y - 1 TO Z - 1
6240  GOSUB 6700
6250  NEXT I
6260  GOSUB 59990: GOSUB 59990: PRINT  CHR$ (13) +  CHR$ (4) + "PR#0": GOTO
     41000
6500  GOSUB 6900: GOSUB 6910
6510 LC = 1: GOSUB 50010: IF W$ <  > "Y" THEN 6530
6520  GOSUB 6920: GOSUB 6800
6530  GOSUB 6930
6540 LC = 1: GOSUB 50010: IF W$ = "Y" THEN D = 1: GOTO 6560
6550 D = 0
6560  GOSUB 6940
6570 LC = 1: GOSUB 50010: IF W$ = "P" THEN C = 1: RETURN
6580  IF W$ <  > "S" THEN  GOSUB 59550: GOSUB 59800: GOSUB 5985: GOTO 657
     0
6590 C = 0: RETURN
6600  PRINT "FORMAT BY FIELD #, NAME, SIZE, TYPE:": PRINT
6610  FOR L = 1 TO A(80): IF B(L - 1) <  > 0 THEN 6660
6620  IF L < 10 THEN  PRINT " ";
6630  PRINT L;":  ";A$(L - 1); SPC( 30 -  LEN (A$(L - 1)));: IF A(L - 1) <
     10 THEN  PRINT " ";
6640  PRINT A(L - 1);" ";: IF A(39 + L) = 1 THEN  PRINT "N": GOTO 6660
6650  PRINT "A"
6660  NEXT L: PRINT : PRINT : RETURN
6700  IF D = 1 THEN 6720
6710  PRINT "( ";I + 1;" )"
6720 E = 1: FOR K = 0 TO A(80) - 1: IF B(K) <  > 0 THEN 6740
6730  PRINT  MID$ (B$(I),E,A(K))
6740 E = E + A(K): NEXT K: PRINT
6750  IF C <  > 0 THEN  RETURN
6760  GOSUB 59990: PRINT "PRESS 'RETURN' TO TERMINATE....": PRINT "OR": PRINT
     "PRESS ANY OTHER KEY TO CONTINUE...."
6770  GET W$: IF W$ =  CHR$ (13) THEN I = Z - 1: RETURN
6780  GOSUB 59980: RETURN
6800  PRINT "USE FIELD NUMBERS (NOT NAMES)."
6810  PRINT "END YOUR LIST BY INPUTTING THE # 0.": PRINT : PRINT
6820 LC = 2:LD = 1: GOSUB 51010: IF W < 0 OR W > A(80) THEN  GOSUB 59560: GOSUB
     59800: GOSUB 59850: GOTO 6820
6830  IF W = 0 THEN  RETURN
6840 B(W - 1) = 1: HTAB (1): CALL  - 868: GOTO 6820
6900  FOR I = 0 TO 39:B(I) = 0: NEXT I: RETURN
6910  GOSUB 59980: PRINT "WANT TO SUPPRESS ANY FIELDS (Y/N)?": PRINT : RETURN

6920  GOSUB 59980: PRINT "INPUT FIELD(S) TO BE SUPPRESSED.": RETURN
6930  GOSUB 59980: PRINT "SUPPRESS RECORD NUMBERS (Y/N)?": PRINT : RETURN

6940  GOSUB 59980: PRINT "OUTPUT TO SCREEN OR PRINTER (S/P)?": PRINT : RETURN

7000  GOSUB 5800: IF G = 1 THEN G = 0: GOTO 41000
7100  GOSUB 59980: PRINT  TAB( 9);"--- FIND   RECORD(S) ---": GOSUB 5900
7110  GOSUB 5700: IF G = 1 THEN G = 0: GOTO 41000
7200  GOSUB 6500
7210  GOSUB 59980: PRINT "SEARCH FIELD NAMED?": PRINT
7220 LC = 10: GOSUB 50010
7230  GOSUB 7900: IF G = 1 THEN G = 0: GOTO 7210
7300  GOSUB 59990: PRINT "LOOK FOR A MATCH WITH?": PRINT
7310 LC = A(B): GOSUB 50010: GOSUB 59990
7320 F = 1: IF B > 0 THEN  FOR I = 0 TO B - 1:F = F + A(I): NEXT I
```

```
7330  GOSUB 59980: IF C = 1 THEN  PRINT  CHR$ (13) +  CHR$ (4) + "PR#1": GOSUB
    59990
7340  PRINT "SEARCH OF FIELD NAMED: ";A$(B): PRINT "IN RECORDS ";Y;" THROU
    GH ";Z
7350  PRINT "FOR THE STRING: ";W$: GOSUB 59990:X$ = W$: GOSUB 66000
7400  FOR I = Y - 1 TO Z - 1
7410 T$ =  MID$ (B$(I),F,A(B))
7420  FOR J = 0 TO A(B) -  LEN (X$)
7430  IF X$ <  >  MID$ (T$,J + 1, LEN (X$)) THEN 7500
7440  GOSUB 67000
7450 J = A(B) -  LEN (X$)
7500  NEXT J: NEXT I
7510  GOSUB 59990: GOSUB 59990: PRINT  CHR$ (13) +  CHR$ (4) + "PR#0": GOTO
    41000
7900 B =  - 1: FOR K = 0 TO A(80) - 1: IF W$ = A$(K) THEN B = K:K = A(80)
7910  NEXT K
7920  IF B >  = 0 THEN  RETURN
7930  GOSUB 59980: PRINT "NO FIELD NAMED: ";W$:I = 500: GOSUB 59970: GOSUB
    59990
7940  PRINT "VALID FIELD NAMES ARE:": PRINT
7950  FOR K = 0 TO A(80) - 1: PRINT A$(K):I = 500: GOSUB 59970: NEXT K
7960 G = 1: RETURN
8000  GOSUB 5800: IF G = 1 THEN G = 0: GOTO 41000
8100  GOSUB 59980: PRINT  TAB( 9);"--- SORT  RECORD(S) ---": GOSUB 5900
8110  GOSUB 5700: IF G = 1 THEN G = 0: GOTO 41000
8120  IF Z - Y < 1 THEN E$ = "YOU MUST PROVIDE A RANGE > 1 TO SORT.": GOSUB
    59800: GOTO 41000
8200  GOSUB 59980: PRINT "SORT ON FIELD NAMED?": PRINT
8210 LC = 10: GOSUB 50010
8220  GOSUB 7900: IF G = 1 THEN G = 0: GOTO 8200
8300  GOSUB 59980: PRINT "SORTING RECORD NUMBERS ";Y;" THROUGH ";Z;
8400 Y = Y - 1:Z = Z - 1:F = 1: IF B > 0 THEN  FOR I = 0 TO B - 1:F = F +
    A(I): NEXT I
8410 J = Z - Y + 1
8420 J =  INT (J / 2)
8430 K = Z - J
8440 D = 0
8450  FOR I = Y TO K
8460 L = I + J
8470  IF A(40 + B) = 1 THEN 8500
8480  IF  MID$ (B$(I),F,A(B)) <  =  MID$ (B$(L),F,A(B)) THEN 8550
8490  GOTO 8510
8500  IF  VAL ( MID$ (B$(I),F,A(B))) <  =  VAL ( MID$ (B$(L),F,A(B))) THEN
    8550
8510 T$ = B$(I)
8520 B$(I) = B$(L)
8530 B$(L) = T$
8540 D = 1
8550  NEXT I
8560  IF D > 0 THEN 8440
8570  IF J > 1 THEN 8420
8580  GOTO 41000
9000  GOSUB 5800: IF G = 1 THEN G = 0: GOTO 41000
9100  GOSUB 59980: PRINT  TAB( 9);"--- TALLY  RECORD(S) ---": GOSUB 5900
9110  GOSUB 5700: IF G = 1 THEN G = 0: GOTO 41000
9200  GOSUB 59980: PRINT "TALLY ON FIELD NAMED?": PRINT
9210 LC = 10: GOSUB 50010
9220  GOSUB 7900: IF G = 1 THEN G = 0: GOTO 9200
9230  IF A(40 + B) <  > 1 THEN E$ = "FIELD SPECIFIED IS NOT NUMERIC.": GOSUB
    59800: GOTO 41000
9300 D = 0:F = 1: IF B > 0 THEN  FOR I = 0 TO B - 1:F = F + A(I): NEXT I
9310  FOR I = Y - 1 TO Z - 1
9320 D = D +  VAL ( MID$ (B$(I),F,A(B)))
9330  NEXT I
9400  GOSUB 59980: PRINT "TALLY FOR FIELD NAMED ";A$(B)
9410  PRINT "IN RECORD(S) ";Y;" THROUGH ";Z;" IS:": GOSUB 59990
9420  PRINT D
9430  GOSUB 59990: PRINT "PRESS ANY KEY TO CONTINUE...."
9440  GET W$: GOTO 41000
10000  GOSUB 59980
```

```
10010  PRINT "DO YOU WISH TO ERASE THIS FILE (Y/N)?": PRINT
10020 LC = 1: GOSUB 50010
10030  IF W$ <  > "Y" THEN 40000
10040  CLEAR : GOSUB 59980: PRINT "FILE HAS BEEN ERASED....":I = 1000: GOSUB
     59970: GOTO 40000
11000  GOSUB 59980: ONERR  GOTO 30900
11010  PRINT  CHR$ (13) +  CHR$ (4) + "CATALOG"
11020  GOSUB 59990: PRINT "PRESS ANY KEY TO RETURN TO MENU....": PRINT
11030  GET W$: POKE 216,0: GOTO 40000
12000  GOSUB 59980: PRINT "RETURNING TO DISK EXECUTIVE DESTROYS": PRINT "A
     NY DATA CURRENTLY IN MEMORY.": GOSUB 59990: PRINT "IS THAT OK (Y/N)?"
     : PRINT
12010 LC = 1: GOSUB 50010: IF W$ <  > "Y" THEN 40000
12020  GOSUB 59980: END
20000  IF AA$ = "D" THEN 10000
20100  GOSUB 59980: PRINT  TAB( 6);"--- READ FILE FROM DISK ---": GOSUB 59
     990
20110  CLEAR : GOSUB 1900: GOSUB 20900: ONERR  GOTO 30900
20200  PRINT X$;"OPEN ";W$
20210  PRINT X$;"READ ";W$
20300  FOR I = 0 TO 83: INPUT A(I): NEXT I
20310  FOR I = 0 TO 39: INPUT A$(I): NEXT I
20320  FOR I = 0 TO A(81): INPUT B$(I): NEXT I
20400  PRINT X$;"CLOSE ";W$
20410 AA$ = "D": POKE 216,0: GOTO 40000
20900  PRINT "SPECIFY FILE NAME (MAX 30 CHARACTERS):": PRINT
20910 LC = 30: GOSUB 50010
20920 X$ =  CHR$ (13) +  CHR$ (4): RETURN
30000  GOSUB 41900: IF G = 1 THEN G = 0: GOTO 40000
30010  GOSUB 5800: IF G = 1 THEN G = 0: GOTO 40000
30100  GOSUB 59980: PRINT  TAB( 7);"--- SAVE FILE TO DISK ---": GOSUB 5999
     0
30110  GOSUB 20900: ONERR  GOTO 30900
30200  PRINT X$;"OPEN ";W$
30210  PRINT X$;"DELETE ";W$
30220  PRINT X$;"OPEN ";W$
30230  PRINT X$;"WRITE ";W$
30300  FOR I = 0 TO 83: PRINT A(I): NEXT I
30310  FOR I = 0 TO 39: PRINT A$(I): NEXT I
30320  FOR I = 0 TO A(81): PRINT B$(I): NEXT I
30400 T$ = "CLOSE " + W$: PRINT X$;T$
30410  POKE 216,0: GOTO 40000
30900  GOSUB 59980: PRINT "I/O ERROR. RECOMMEND CHECKING DISK.": GOSUB 599
     90: PRINT "PRESS ANY KEY TO RETURN TO MENU...."
30910  GET W$: POKE 216,0: GOTO 40000
40000  GOSUB 59980: GOSUB 59990: GOSUB 59990
40010  PRINT  TAB( 10);"1. USE APPLE DATA BASE"
40020  PRINT  TAB( 10);"2. DEFINE RECORD FORMAT"
40030  PRINT  TAB( 10);"3. READ FILE FROM DISK"
40040  PRINT  TAB( 10);"4. WRITE FILE TO DISK"
40050  PRINT  TAB( 10);"5. READ DISKETTE CATALOG"
40060  PRINT  TAB( 10);"6. EXIT TO DISK EXECUTIVE"
40080  PRINT : PRINT
40090  PRINT  TAB( 10);"9. ERASE FILE FROM MEMORY"
40500  VTAB (23): GET I$:I =  ASC (I$) - 48: IF I < 0 OR I > 9 THEN 40500
40510  ON I + 1 GOTO 40500,41000,1000,20000,30000,11000,12000,40500,40500,
     10000
41000  GOSUB 41900: IF G = 1 THEN G = 0: GOTO 40000
41010  GOSUB 59980: GOSUB 59990: GOSUB 59990
41020  PRINT  TAB( 13);"1. APPEND RECORD(S)"
41030  PRINT  TAB( 13);"2. INSERT RECORD(S)"
41040  PRINT  TAB( 13);"3. CHANGE RECORD(S)"
41050  PRINT  TAB( 13);"4. DELETE RECORD(S)"
41060  PRINT  TAB( 13);"5. LIST   RECORD(S)"
41070  PRINT  TAB( 13);"6. FIND   RECORD(S)"
41080  PRINT  TAB( 13);"7. SORT   RECORD(S)"
41090  PRINT  TAB( 13);"8. TALLY  RECORD(S)"
41100  PRINT  TAB( 13);"9. ** NEXT  MENU **"
41500  VTAB (23): GET I$:I =  ASC (I$) - 48: IF I < 0 OR I > 9 THEN 41500
```

```
41510  ON I + 1 GOTO 41500,2000,3000,4000,5000,6000,7000,8000,9000,40000
41900  IF AA$ <  > "D" THEN  GOSUB 59980: PRINT "NO FILE IN MEMORY....":I =
     500: GOSUB 59970:G = 1
41910  RETURN
50000 LC = 40
50010 W$ = ""
50020  GET I$: IF  ASC (I$) <  > 8 THEN 50060
50030  IF  LEN (W$) > 1 THEN W$ =  LEFT$ (W$, LEN (W$) - 1): PRINT I$;: CALL
     - 868: GOTO 50020
50040  IF  LEN (W$) = 1 THEN W$ = "": PRINT I$;: CALL  - 868
50050  GOTO 50020
50060  IF  ASC (I$) = 44 THEN  GOSUB 59500: GOSUB 59800: GOTO 50020
50070  IF  ASC (I$) <  > 13 THEN 50100
50080  IF  LEN (W$) > 0 THEN  RETURN
50090  GOSUB 59520: GOSUB 59800: GOTO 50020
50100  IF  LEN (W$) >  = LC THEN  GOSUB 59510: GOSUB 59800: GOTO 50020
50110 W$ = W$ + I$: PRINT I$;: GOTO 50020
51000 LC = 40:LD = 0
51010 W = 0
51020  GOSUB 50010
51030  IF  LEFT$ (W$,1) =  CHR$ (0) THEN 51060
51040  IF ( LEFT$ (W$,1) = "." OR  LEFT$ (W$,1) = "-") AND ( LEN (W$) > 1)
     THEN 51070
51050  IF  LEFT$ (W$,1) > "/" AND  LEFT$ (W$,1) < ":" THEN 51070
51060  GOSUB 59530: GOSUB 59800: GOSUB 59850: GOTO 51010
51070 LI =  LEN (W$):W =  VAL (W$):W$ =  STR$ (W): IF LI <  > LEN (W$) THEN
     51060
51080  IF LD = 0 THEN  RETURN
51090  IF (W -  INT (W)) <  > 0 THEN  GOSUB 59540: GOSUB 59800: GOSUB 5985
     0: GOTO 51010
51100 LD = 0: RETURN
59500 E$ = "SORRY, COMMAS NOT PERMITTED!": RETURN
59510 E$ = "MAX FIELD LENGTH HAS BEEN REACHED!": RETURN
59520 E$ = "PLEASE! YOU MUST INPUT SOMETHING!": RETURN
59530 E$ = "INVALID NUMERICAL FORM. RE-ENTER DATA.": RETURN
59540 E$ = "AN INTEGER VALUE IS REQUESTED, PLEASE.": RETURN
59550 E$ = "PLEASE RESPOND IN THE FORMAT REQUESTED.": RETURN
59560 E$ = "NUMBER OUTSIDE VALID RANGE. RE-ENTER.": RETURN
59800 CH =  PEEK (36):CV =  PEEK (37): VTAB (22): HTAB (1)
59810  GOSUB 59950
59820  PRINT E$;
59830 I = 500: GOSUB 59970
59840  POKE 36,0: CALL  - 868: VTAB (CV + 1): HTAB (CH + 1): RETURN
59850  HTAB (1): CALL  - 868: RETURN
59950  CALL  - 198: FOR I = 1 TO 30:I = I: NEXT I: CALL  - 198: RETURN
59970  FOR I = I TO 0 STEP  - 1:I = I: NEXT I: RETURN
59980  HOME
59990  PRINT : PRINT : PRINT : RETURN
```

# DATA BASE APPLICATIONS

In this section, I will present and discuss in detail a number of practical applications of the data base management program provided in this book. Methods of capitalizing on its strengths and minimizing the effects of its weaknesses will be pointed out. (And, later in the book, those that really want to delve into the art and science of programming, will be provided with tips on building in capabilities to suit their own special requirements!)

## MAILING LISTS

### The Classic Application

Perhaps the most common use for a data base management program is to maintain and utilize a mailing list. Virtually everyone who has the savvy to utilize a personal computer has more than enough personal contacts to justify using a computer to help keep track of friends, acquaintances, and associates. It can be a lot more effective, in many instances, than keeping an address book. (Except, of course, for your *very* best friends. The whereabouts of those people should always be kept in your head and heart. Nothing that sacred should be trusted to *any* computer!)

### Defining the Record Format

Any use of a computer requires tradeoffs. A tradeoff one faces when defining the fields to be used in an application, such as a mailing list, revolves around how much room to assign to each field. Remember, you must tell the program the *maximum* number of characters that will be allowed in each field.

Now making this decision can be a little difficult in an application such as this *because you might not know what names you are likely to*

*encounter in the future!* So, how on earth can you assign a maximum length to the field that will hold the names of persons on your mailing list? Alas, readers, here is a typical practical case where nothing beats *experience* (or some hard scientific evidence).

If you have been "managing" the list of names that will go in your data base for some time, chances are good you are familiar with some of the longer names in your list. (They often stick out like sore thumbs.) If so, count the number of characters in the longest names, including spaces, initials, and surnames, if applicable. You might also want to decide whether you will include titles or their abbreviations (such as Jr.) in the "Name" field. To be on the safe side for possible future additions to your list, you might want to add a few extra character positions to your final tally. Now take the highest number of characters you come up with and make that your field length assignment.

But wait, what is the tradeoff in this situation? Well, the longer you permit each field to be, the longer each record will be. As the lengths of records become longer, the number of records that can be stored in a file becomes less. That is the tradeoff. Suppose your computer has 18,ØØØ bytes of memory left after the data base program has been loaded. If you define a name field to be 3Ø characters in length, when it could do the job with just 2Ø, you are going to waste a lot of memory. For instance, a record length of 1ØØ characters would mean you could store 18Ø records in a file of 18,ØØØ bytes. If each record in that file only contained 9Ø characters, then up to 2ØØ records could be stored in the file.

So, in many cases, it pays to closely evaluate such points as whether it might be worthwhile to, say, shorten surnames to initials and abbreviate all titles, thus enabling this maximum field length to be reduced.

Having said all that, I am going to tell you that, in my own personal experience having scanned upward of half a million names in the past 1Ø years, I have seldom found it necessary to allocate more than 2Ø characters to a "Name" field. My uses for mailing lists have usually been commercial. In such applications, it is generally acceptable to use initials for all but the last name and abbreviate titles such as Junior and Senior. Thus, the "Name" field in this illustration is set at 2Ø characters.

You have the same kind of decision to make for the lengths of all the fields in a record. For illustrative purposes, the "Address" field in this example will be set at 3Ø characters. This generally provides plenty of room for a street address, including an apartment number, provided judicious use of abbreviations is made for such details as the road, street, avenue, circle, plaza, or whatever. But I can also tell you from experience that if you want to print addresses on labels that are only 2¾ inches wide, then you had better limit the number of characters (at 1Ø to the inch on standard printers) in the "Address" field to a maximum of 27!

The point has been made: select a field length that will provide room for your typical entries, while judiciously limiting the size to avoid

wasting memory space! In this practical illustration, the length for the "City" field has been set at 12 characters. The "State" field has been limited to two characters (using standard post office abbreviations). And the "Zip Code" field has been set at five positions, forgetting for the moment the urgings of the USPS for conversion to the "ZIP + 4" format.

### A Zip Code Is Not a Number

At least not when using this data base management program. You should consider it as an alphanumeric field for the following reason: Fields that are declared to be of the "numeric" type can only contain "significant" digits. Thus, a valid zip code, such as Ø3456, could only be entered into a numeric field as 3456. The post office would frown upon (indeed, outright reject) any letter that entered its mechanized system with only four digits in the zip code. For it has been decreed that all zip codes shall contain five digits (or five plus four!). Verily, I say unto you, tell the data base management program that the zip code field is *alphanumeric.*

I have defined a field for telephone numbers. It too is declared to be an alphanumeric field so that I can use characters such as parentheses. Standard telephone numbers in the United States can generally be stored within 14 character positions in the form: (123) 456-789Ø.

Also, for this application I have created two other "additional information" fields. One is labeled "Category" and the other "Code" for the sake of discussion.

I have arbitrarily, for illustrative purposes, created six different categories into which I will group people in this mailing list. They are: customer, finance, legal, personal, prospect, and supplier.

Under the "Code" field I will have "subcategories" that are different for each "category" along the following lines: Under "customer" I have the "codes" GENL (general), MTNC (maintenance), and SPCL (special). Under "finance" I have ACNT (accountant) and GOVT (government). Under "legal" I have TAX (obviously for assistance from qualified experts), GEN (for general counsel), CIVL (for civil matters), and BUS (for attorneys specializing in business contracts). Finally, under "prospect" are INIT (for initiate contact), FLUP (for follow-up), and GETM (for time to close the sale).

Thus, as I enter each personal contact into the file, I will use the "Category" and "Code" fields to categorize and code (within that category) each addressee.

I have limited the "Category" field to a maximum of eight characters and the "Code" field to a maximum of four positions. Through the judicious use of abbreviations in the "Code" field, I can create many subgroups without using much storage space.

There are eight defined fields in this example application. If you add up all the characters assigned to each field, you have the record

length. It is 95 characters in this example. If your computer has 18,ØØØ bytes of memory left when the data base program is installed, then you would be able to store up to 19Ø records in a single file using this record format. (If you have a 48K Apple II with Applesoft in ROM, you will likely have a lot more than 18,ØØØ bytes of memory available for a data file. On the other hand, if you are running Applesoft in RAM, you may have less than 18,ØØØ bytes available for a file.)

You would enter all the record-formatting directives just described by selecting Primary Menu option number 2 (DEFINE RECORD FORMAT), then responding to the prompts provided by the program. These prompts would be for the name, maximum length, and type—(A)lphanumeric or (N)umeric—for each field. Here is a summary of the field definitions just described:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:
 1:  NAME                          2Ø A
 2:  STREET                        3Ø A
 3:  CITY                          12 A
 4:  STATE                          2 A
 5:  ZIP CODE                       5 A
 6:  TELEPHONE                     14 A
 7:  CATEGORY                       8 A
 8:  CODE                           4 A
```

## APPEND Your Data

After you have defined the fields and returned to the Primary Menu, you select option number 1: USE APPLE DATA BASE to move to the Operations Menu. Once there, you can start building up a file using the APPEND option.

The field format presented here is strictly for illustrative purposes. You, quite likely, would modify it to suit your own needs. However, for illustrative purposes, I am going to hypothesize that a number of records, including the four shown here, have been keyed into the current memory-resident file:

```
( 1 )
JOHN DOE
123 PLEASANT STREET
ANYTOWN
NJ
Ø7777
(2Ø1) 999-1234
PROSPECT
INIT

( 2 )
BILL SMITH
5 STAR AVENUE
BRIGHTON
CA
9Ø186
```

```
(213) 555-1234
PROSPECT
GETM

( 3 )
WAYNE SWEETOOTH
10518 WAFFLE BLVD.
SYRUPVILLE
NY
10101
(212) 666-1234
CUSTOMER
SPCL

( 4 )
ALICE PAXTON
101 ROUTE 5 - APT. 3B
PETERSON
NH
03030
(603) 111-1234
LEGAL
TAX
```

## Doing a Mailing

Now suppose you had 150 records such as those in your file. You decide that you would like to have a neatly organized "master" list to use as your quick-reference "address book."

Fine, the first operation you might perform could be to perform a sort on the Zip Code field. After selecting choice 7: SORT RECORD(S) from the Operations Menu, you would respond appropriately to the program prompts so as to have the entire file sorted. You would specify that the sort be performed on the field named Zip Code.

At the completion of the sort, you could have the newly organized list outputted to your system's printer using the LIST option. By specifying that the record numbers be displayed, you would end up with a "master" list that would be useful when updating the file. This is because it would show the "reference" position of each record within the sorted file. This list would be arranged in zip code order. Thus, knowledge of a person's zip code would quickly allow you to locate the other specifics pertaining to that individual, such as address, phone number, category, and so forth.

Suppose, next, that you wanted to mail a circular to all of the "prospects" on your list. One way to approach the task would be to select the FIND option. You could specify that the entire file be searched, examining the "Category" field for occurrences of the word "prospect."

Furthermore, you could set up the output so that ready-to-mail labels were produced on a printer! How? By simply directing that field numbers 6, 7, and 8 be suppressed! That would mean that only fields 1 through 5 (Name, Address, City, State, and Zip Code) would be printed. Just what you want on a mailing label.

Perhaps, after you have sent out your circulars or flyers, you decide that it would be a good idea to pay a visit to some of your prospects. Which ones? Why, those residing in your own and adjoining states. You could get organized for this tour by sorting your list according to the contents of the "State" field. Once this had been done, you could obtain subgroups within states by sorting on the contents of the "Category" field. You could follow this up by organizing the "prospect" subgroup according to the contents of the "Code" field. This would group your prospects into those that had never been visited (INIT), those that needed follow-up (FLUP), and those that were ready to be asked to sign on the dotted line (GETM).

Now you could efficiently plan your trip to cover local states, visiting only the subgroup of potential clients that needed your immediate personal attention.

Of course, if you didn't want to actually visit prospects, you might think it beneficial to conduct a telephone sales campaign. OK, just press a few more keys on your computer. Tell the data base program that you want to FIND all "prospects" in the "Category" field. Set up the output (by suppressing field numbers 2, 3, 4, 5, and 7) so that you obtain a list that only shows the contact's name, telephone number, and code. In a few minutes you have precisely the list you need for the job!

The FIND option just described would give you the list you wanted. It should be pointed out, however, that if you already had the file ordered by sorting it on the "Category" field, you could obtain the list of prospects faster using an alternate method. You could direct it to list just the records that contain the entry "prospects" in the "Category" field. The actual range of record numbers to be listed could be obtained from your previously sorted "master" list.

### Update Your File as Necessary

Naturally, once you have established your initial file, you will store it for safekeeping on a diskette. Anytime you need to use it again, you just access the same diskette using option 3 of the Primary Menu. Once the file is back in memory, you can use the APPEND, INSERT, DELETE, or CHANGE options provided by the Operations Menu to update your information.

You will likely find the CHANGE operation particularly useful in maintaining a mailing list. Remember, you use it to change the contents of individual fields within records. So, when a person moves to a new address, you can just alter the "Street," "City," "State," and "Zip Code" fields while leaving all the other fields intact. Or, if a prospect is upgraded to a customer, then you just make the appropriate alteration to the "Category" field.

Remember, too, if you have your file ordered, say, by zip code, you can often keep it in order using the INSERT operation. If a new entry needs to be added, you can just refer to your master list to find the proper

point in the file at which to make the insertion. Since sorting a lengthy file can take some time, this technique is often wise if you just want to make a few additions without disturbing the ordering of the records already present.

### Experiment, But Remember the Limits

Of course, the record format illustrated here is merely to give you a starting point. You have the freedom to create whatever fields you desire. Perhaps you want to provide extra fields in each record for adding notes and miscellaneous remarks over time. Fine. You can define up to 40 fields for a single record. Remember, however, that no field can be longer than 40 characters. And, that all of the fields together may not exceed 236 characters. Stay within those limits and "the program is your oyster."

## INDEXES

If you have ever had to prepare an index for a book or manual, you know what a chore it can be. The classic manual method is to start the project with a large stack of three-by-five-inch file cards. Then you go through the text to be indexed and extract the key words. Perhaps, as you work, you will try to keep the cards in alphabetical order. Or perhaps you will wait until you have a huge stack of cards, then spend several hours putting it in order. The next step is to type all the information on your cards into a neatly arranged index.

If you have never had to create an index for a publication, consider yourself fortunate. You can easily spend the better part of a day or two just trying to prepare an index for a book. Few people find this kind of tedious task enjoyable.

But a data base management program such as this can really cut an indexing job down to size. This application of the program is so simple, yet saves so much work, that you might very well consider it miraculous.

### All It Takes Is Two Fields

Sure, the typical indexing chore only requires constructing records that have two fields. One for holding words. The other for storing their corresponding page location(s).

In the example illustrated here, I named the field that holds the words as the Word field, limited it to a maximum of 15 characters and specified that it be alphanumeric in type. The associated field for storing the page number where the word appeared was named the Page field. I limited it to four digit positions. I designated it as numeric in type. I did

this because I knew it would be useful, in certain instances, to arrange repeated instances of the same word according to page number.

Once the indexing file has been formatted, it is a simple matter to go through the text and input the relevant data into the computer. Each time a word is spotted that needs to be included in the index, it is entered in the Word field. The page number on which it occurs is then entered in the Page field. If the same word occurs in different contexts, fine! Enter it again.

Here is what the raw data from a typical indexing project might look like when initially listed by the data base management program:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1: WORD                                15 A
 2: PAGE                                 4 N
EXPENDITURES
128

EXPENDITURES
355

EXPENDITURES
412

CORPORATIONS
11

CORPORATIONS
225

BIOLOGICAL
45

MOTOR VEHICLES
172

RAILROADS
419

RAILROADS
150

TELEPHONES
212

TOYS
145

WATER
17

WATER
331

HOUSING
50

EDUCATION
287
```

Note that no attempt was made to alphabetize words as they were found when reading the text. Words of importance were simply inputted as found. Note, too, that in this example the text was perused at random. (This was done in order to illustrate a capability of the program. Typically, you would probably construct an index by reading through a text in a linear fashion. However, as far as the data base management program is concerned, it doesn't make any difference how you go about acquiring the raw data that you plan to use in the index.)

### The Computer Does the Hard Work

Sure it does; that is what you have it for, right? Once the raw data for the index has been inputted, all you need do is tell the program to sort on the Word field. Here is how the example index would appear following a sort of the entire file on that field:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  WORD                                15 A
 2:  PAGE                                 4 N

( 1 )
BIOLOGICAL
45

( 2 )
CORPORATIONS
11

( 3 )
CORPORATIONS
225

( 4 )
EDUCATION
287

( 5 )
EXPENDITURES
355

( 6 )
EXPENDITURES
412

( 7 )
EXPENDITURES
128

( 8 )
HOUSING
5Ø

( 9 )
MOTOR VEHICLES
172
```

```
( 10 )
RAILROADS
150

( 11 )
RAILROADS
419

( 12 )
TELEPHONES
212

( 13 )
TOYS
145

( 14 )
WATER
331

( 15 )
WATER
17
```

Now you have the list alphabetized. If the index was large and you had numerous entries under the same word, you might want to have the computer go one step further.

Note, for instance, that the word "expenditures" has three entries in the sample listing. However, because of the manner in which the entries occurred and the way the sort on the Word field was performed, the page sequence is not in ascending order. You could now perform a sort on the Page field, over just those records (numbers 5 through 7 in the example) that contained the word "expenditures." In just a few moments those records would be rearranged as:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  WORD                              15 A
 2:  PAGE                               4 N

( 5 )
EXPENDITURES
128

( 6 )
EXPENDITURES
355

( 7 )
EXPENDITURES
412
```

Now those words are also ordered sequentially by numerical (page) order of appearance!

Naturally, if you had a number of repeated occurrences, you would perform a sort on each subgroup as necessary.

Once this had been accomplished, you would be able to re-list the entire finished index, arranged both alphabetically and by order of page appearance.

Sure beats the old file card method. In my experience, it wins by a factor of three or four in terms of ease and speed. Personally, *I like that!*

## HOUSEHOLD INVENTORIES

Have you any idea of what all of your household possessions are worth? Could you, following a disaster such as a fire, make an accurate list of what was in your home?

If you are paying for insurance, *and you do not have such a list,* then you are literally likely to be throwing your money away. Because, when it comes time to collect, you are going to have to demonstrate, in a reasonable manner, that you knew just exactly what your losses were.

One way to solve the potential problem is to stash all your receipts for major household purchases in a safe deposit box at a bank.

Another way is to use your computer to maintain an up-to-date list of all important valuables. Then keep a copy of that list (perhaps even a backup diskette!) in a safe deposit box. There are other advantages to following this method, as will be pointed out.

### You Need Not Be Elaborate

Using a data base management program, it doesn't take much work to come up with a neat and efficient system for maintaining an inventory of household belongings. I use a record format having only six fields. Perhaps you would like to duplicate the format I have found sufficient. But, you can add or delete fields to suit yourself. Here is the record structure I use:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  ITEM                    40 A
 2:  LOCATION                10 A
 3:  COST ($)                 6 N
 4:  DATE ACQD                8 N
 5:  NOTE 1                  40 A
 6:  NOTE 2                  40 A
```

I like to leave plenty of room in the first (Item) field to adequately describe things. Although it is seldom necessary to use it all, it is nice to have it available for those special cases.

The "Location" field is only assigned 10 character positions. It is a good idea to develop your own set of abbreviations to use in this field. For

instance, LR for living room, MBR for master bedroom, etc. But 1Ø characters gives you the space to spell out kitchen, bedroom, garage, attic, and so on, if that suits you better. Be consistent in your assignments or abbreviations. Doing so will enable you to perform meaningful sorts on the "Location" field so that you can group the contents of each room.

The "Cost" field is shown with a dollar sign in parentheses as a reminder that amounts are entered to the nearest whole dollar. This field is declared to be strictly numeric so that the program's TALLY operation can be used. This can be beneficial as will be shown later. I limit the length of this field to just six digits because I don't have any individual items worth more than $999,000. Of course, if you belong to the multimillionaire set, then you can always set this field larger. (But remember the Apple II can only use nine significant digits. Don't waste space by allocating too many characters to this field.)

Now the "Date Acquired" field (abbreviated DATE ACQD to fit within the 1Ø characters allowed for a field name) needs special mention. I have made it numeric in type and set it to a length of eight characters. I have done this so that I can use a little scheme that permits data to be numerically arranged by date. The scheme is to encode all dates in the form "year-month-day." Thus, the date May 25, 1982, would be entered as: 1982Ø525. Note that all four digits are used for the year. I do this because I am an optimist. You could reduce this field to just six digits and store the year as just two digits. Thus, the same date could be stored as 82Ø525. But, gee whiz, what happens when the year 2ØØØ rolls around and you want to arrange your records in chronological order? (See what I mean about being an optimist? I am planning on my little old Apple lasting another 17 years!)

Some of you may not want to bother with the last two fields. They are maximum length fields for keeping miscellaneous notes about the items recorded. I'll show you the kind of information you might keep track of in a couple of sample entries. Then you can decide for yourself whether you want to assign more or less fields and/or characters-per-field for this purpose.

Remember the old tradeoff. The more room each record takes, the less records you can have in a file. It turns out, on my system, that I can store 147 records in a file with the format shown. Now I don't have anywhere near 147 items worthy of recording in my household, so the record length is just fine for my purposes. But, your situation may be different. If you will need more records in a file, then shorten or eliminate fields 5 and 6. (Of course, you *could* create two or more files for holding all your household inventory records, if necessary.)

### Fill the File

Once the file format has been established, you just start going around your house or apartment, eyeballing everything of value. If it's worth keeping track of (that means getting reimbursed if the place goes down

the tubes), then you make an entry in your data base. Here are some typical examples of entries you might make in such a file:

```
ITEM: SOFA
LOCATION: LR
COST ($): 840
DATE ACQD: 19780214
NOTE 1: LOFT'S FURNITURE STORE.
NOTE 2: TREAT FABRIC EVERY 5 YEARS.

ITEM: REFRIGERATOR (G.E.)
LOCATION: KITCHEN
COST ($): 917
DATE ACQD: 19820506
NOTE 1: FRIENDLY APPLIANCES INC.
NOTE 2: 4 YEAR WARRANTY ON MOTOR.

ITEM: PEARL NECKLACE
LOCATION: SAFE
COST ($): 1200
DATE ACQD: 19801224
NOTE 1: FINEST JEWELERS CO.
NOTE 2: INSURED BY RIDER. RENEW ANNUALLY.
```

OK, you get the idea.

### Evaluating Your Assets

Once you have inputted all the data, you can begin to take a look at your situation. You may discover some surprises.

For example, perhaps the first experiment to try is to simply perform the TALLY operation. Try it on the Cost ($) field, over the entire file. Are you startled by the figure it comes up with? Do you feel you are adequately insured?

There are all kinds of other ways to examine your situation in a realistic light. Have you ever wondered which room to start hauling stuff out of if someone told you that you had to evacuate your residence within two hours? Well, if you want to get an idea of the most valuable room in your domicile, do the following:

First sort the data file on the contents of the Location field. This will group all of the articles in each room (provided you have been consistent with your room names and abbreviations). Then you can perform the TALLY operation over the range of records residing in each group. In a matter of minutes you can come up with the total value of your possessions in each room. Any surprises?

Did you know that, if you should ever suffer a catastrophe, the insurance company is going to reimburse you for your losses on a depreciated schedule (unless you have "replacement value" coverage)? You can get a pretty good idea of how "aged" your overall household goods are by doing the following: Sort the file by the contents of the Date

Acquired field. If you used the dating technique I suggested earlier, the records will then be arranged according to the date acquired. Your oldest items will head the list.

With the file organized in this fashion, the TALLY option can be used to obtain sums (in the Cost ($) field) over the range of records spanning each calendar year. Then, if you are the financial-analyst type, you can quickly get an idea of your net worth by applying an appropriate depreciation formula to the sum obtained for each annual period.

You might also be able to use the sum obtained for the current year to figure the applicable sales tax deductions that could be applied against your income taxes. Wouldn't that be pleasurable?

### Make a Few Copies for the Safe Deposit Box

You bet. Use the LIST option to print a few copies of the file for storage in a safe location. You might even decide to keep a copy of the file on diskette in a special, safe place. Why be sorry? Make that computer provide you with some security! It owes it to you. After all, you keep it supplied with juice!

## TAX DEDUCTIONS

One reason people dislike paying personal income taxes is because of all the work it takes to document deductions. There are two ways you can go. Take the "easy" way and settle for the "standard deduction" allowed by the government. Or take the "hard" way and prove that your deductions exceeded the "standard."

Some people never bother to even examine the second alternative because they don't want to be bothered keeping the necessary documents and records, to say nothing of then doing the appropriate calculations.

But, the use of your data base management program can make the task of tracking and calculating legitimate tax deductions a lot easier. Maybe easy enough for you to exercise your right to pay only your fair share of taxes. The proper fair share can only be made by accurately examining your options. Why cheat yourself?

### You Still Have to Keep Records

No, the use of this program will not eliminate the need to keep receipts of all your tax-deductible expenses. But it can make it a lot easier to keep track of those expenses, and it can help you with the critical task of calculating whether your deductions exceed the "standard deduction."

To do this, you will want to establish a data file with records having fields such as what follows on the top of the next page.

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  DATE                    4 N
 2:  DEDUCTION              30 A
 3:  AMOUNT                  8 N
 4:  CATEGORY               10 A
 5:  PAID BY                20 A
 6:  REFERENCE              20 A
```

Note that the Date field is defined as numeric in type and has room for four digits. All dates in this example will be stored as month and day-of-month (i.e., April 1st would be recorded as 401; November 15th as 1115, etc.). It is assumed that the contents of any particular file will be for a specific calendar year period. (If this is not the case in your application, review the preceding application for an example of another way to store dates.)

The Deduction field is used to describe the nature of the deductible expense. Thirty characters should be sufficient for most users, but feel free to size it to your needs.

The Amount field is numeric and allotted eight digit positions. Since, in this application, figures may be given in dollars and cents, one of those positions may be used for a decimal point. Allowing for the use of another position for a minus sign (in case you wanted to record negative values, which might be the case if you needed to adjust a category), then negative amounts of up to −9,999.99 could be stored. Since positive values do not use a sign position (because it is implied), positive values of up to 99,999.99 could be recorded.

The fourth field, assigned a maximum of 10 characters, is used to permit placing expenses into various categories, such as medical, interest, casualty losses, and so forth.

Two more fields, each having 20 characters, allow the recording of additional information. Such as to how the expense was paid and additional document-supporting information.

### Example Entries

Perhaps the best way to get an idea of the practical use of this application is to provide a few typical entries in a data file:

```
DATE: 401
DEDUCTION: DR. JIM DANDY
AMOUNT: 40
CATEGORY: MEDICAL
PAID BY: PERSONAL CHECK # 412
REFERENCE: MARCH BILLING

DATE: 421
DEDUCTION: SALES SEMINAR
AMOUNT: 195
CATEGORY: EDUCATION
```

```
PAID BY: MASTERCARD
REFERENCE: CHARGE NR. 1414

DATE: 430
DEDUCTION: DONOVAN'S PHARMACY
AMOUNT: 12.95
CATEGORY: MEDICAL
PAID BY: CASH
REFERENCE: PRESCRIPTION # 7788
```

Pretty straightforward, right?

### Organize and Deduct!

If you didn't already know it, you are probably rapidly learning that the hardest part about using a data base management program is entering the data! Once you have the data you want to keep track of entered, getting the computer to organize it and extract meaningful results is easy.

For instance, perhaps one of the first directives you will give to a tax deductions data base is to sort the file according to the contents of the "Category" field.

Then you will be in a position to effectively use the TALLY operation. You could, of course, do a summation on the "Amount" field for all of the records in your file. That cannot only satisfy your immediate curiosity, but it can indicate whether it is worth analyzing any further. After all, if the total possible itemized deductions does not exceed the "standard" that anybody can take, then you have all the information you need! If it does, and you plan on itemizing, then you will need to fill in the long tax forms according to categories.

So, you direct the program to tally the various subgroups you have identified in the "Category" field. Remember, you just specify the range of record numbers over which the tally (on the "Amount" field) is to be done. Record each subtotal in the appropriate place on your tax form(s). You can't beat that for convenience, can you?

You might also want to use your tax deductions data base to gather some other kinds of information on your spending. For instance, you could re-sort the file by the contents of the "Paid By" field. Now you would be able to analyze what percentage of your expenses you were paying by credit card, personal check, or other means.

Or, perhaps you would like to view how expenses piled up over the year on a chronological basis. OK, just arrange things in that order by sorting on the "Date" field. Perhaps you will want to make a list on your printer of the file arranged in this order, for future reference.

And, the next time that auditor calls you in for a friendly little chat, you will know that you have the kind of records that can eliminate a lot of arguments.

## SALES ANALYSES

Many people who run a small business need to meticulously keep track of their inventory turnover. The rate of turnover and the margin between acquisition and the actual retail price at which goods are sold determine whether one makes any money. The rate of turnover is an especially significant factor that can often make the difference between success and failure.

For instance, a product with a margin of a mere 10% that turns over six times a year is a lot more profitable than one with a 40% margin that takes a year to be sold.

The fact of the matter is that many small businesspeople do not (cannot?) accurately track their actual turnover rate. The paperwork involved is generally just too time consuming.

But that can be changed with the help of a data base management program. Here is how a data file might be formatted to help perform sales analysis for a small business, such as an antiques dealer.

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  ITEM                    36 A
 2:  CODE                    12 A
 3:  COST                     8 N
 4:  DATE ACQD                8 N
 5:  SOLD FOR                 8 N
 6:  DATE SOLD                8 N
```

The first field is used to store a physical description of the item being inventoried. Another field is assigned for a stock code or other identifying information. These first two fields are alphanumeric in type.

The remaining four fields are designated as numeric in type. The third field is used to record the purchase or wholesale price of an object. The fourth field is used to record the purchase date. This is an important piece of information if one wants to be able to obtain turnover rates. The date is kept in the year-month-day format suggested in an earlier example application. (Thus, January 2, 1982, would be entered as 19820102.) The fifth field holds the price at which an item was actually sold. The last field in a record is for the selling date. This information is needed to keep track of how long an item was stocked before it sold.

Only 80 characters are used in a record, so most Apple systems could store several hundred items in one file. Of course, you can modify the format and field-length assignments as you deem appropriate for your specific case.

### Typical Data

Here are how a few records might look for a businessperson dealing in antiques (see top of next page):

```
ITEM: BRASS BED
CODE: FURN-001-005
COST: 225
DATE ACQD: 19800515
SOLD FOR: 380
DATE SOLD: 19820312

ITEM: FIREPLACE SET
CODE: UTIL-004-012
COST: 42
DATE ACQD: 19811106
SOLD FOR: 85
DATE SOLD: 19820203

ITEM: DOLL HOUSE
CODE: TOYS-003-002
COST: 125
DATE ACQD: 19820614
SOLD FOR:
DATE SOLD:
```

Note that, in the third record, there isn't any entry for the last two fields. This is how the record would look when an item had not yet been sold.

### Sales Analysis

Of course, you build up your initial data file by entering information in the first four fields of a record as items are acquired. Later, when an item is sold, you can locate its record number using the FIND option. You can search on either the "Item" or "Code" field. Once it has been located, you can use the CHANGE option to update the last two fields (actual selling price and the date of the sale).

If you want to know whether you are spending more than you are taking in (whether inventory is increasing), you can perform the TALLY operation on the "Cost" field and then on the "Sold For" field.

A chronological ordering of the file can be obtained by sorting on the "Date Acquired" field. You can then compare the acquisition and selling dates on an item-by-item basis to get an idea of what types of items are selling fastest.

You can review subtotals of various groups of items by first sorting on, for instance, the "Code" field. Then, taking a tally over the range of record numbers comprising each group of interest.

One more little tip: If you want to keep your account current, it is easy to eliminate your old records, *limiting your deletions to only those items that have been sold.* First arrange the file in chronological order by sorting on the "Date Sold" field. Then list the file to ascertain the range of numbers of those records that are obsolete. Use the DELETE option to erase that group of records. Now you have more room to input current items. Doing this on a regular basis will keep your information file current and relevant. (Don't forget, if you want to keep a printed record

of those items that you delete from your file, then just LIST that portion of the file *before* you perform the DELETE operation!)

Small businesspeople need all the help they can get. Could you envision a data base management program helping you when applied as outlined here?

## PARTS LISTS

Engineers and product designers are often plagued with the responsibility of maintaining accurate parts lists. That is, whenever they design a machine, they must methodically itemize each and every part used in the device. This mundane task, while vital, is often shunned by designers because it is not directly related to the creative "design" process.

But, the use of a data base management program can make the upkeep of such a list somewhat less tedious. Here are some of the fields one might want to assign to records that will be used to maintain a product parts list:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  PART                40 A
 2:  QUANTITY             4 N
 3:  STOCK #              8 A
 4:  UNIT COST            6 N
 5:  TOTAL COST           8 N
 6:  VENDOR              20 A
 7:  ADDRESS             30 A
 8:  CITY                12 A
 9:  STATE                2 A
10:  ZIP CODE             5 A
11:  CONTACT             20 A
12:  TELEPHONE           14 A
```

There are fields for storing a description of each part, the quantity of that part used in the device, an in-house stock number, the single unit cost, and the total cost. The total cost would simply be the quantity of parts used times the single unit cost.

The remaining fields are used for keeping information about the primary vendor from which the parts are obtained. Note that these fields are arranged similar to that shown for a mailing list application. Thus, *virtually all of the capabilities described for that application are automatically a part of this application.* Therefore, all of the possible benefits one might derive from being able to organize and manipulate the address-related fields will not be repeated here.

Rather, I will discuss some of the operations one might wish to perform on the parts-related fields.

You can, of course, arrange the parts list into alphabetical order by sorting on the "Part" field. This makes it easy to review what parts have been used in the machine under development.

Using the TALLY operation on the "Quantity" field quickly yields how many items have been used in the device. And, the total parts cost can be obtained by tallying the "Total Cost" field.

Does the unit use a number of different sized, but similar, parts? For instance, perhaps it has six different varieties of nuts and bolts. If these are entered in consistent fashion in the "Part" field, such as:

```
BOLT - 1/4 INCH
BOLT - 3/8 INCH
BOLT - 1/2 INCH
NUT - 1/4 INCH
NUT - 3/8 INCH
NUT - 1/2 INCH
```

then the data base program can be used to group similar parts. Once grouped, tally operations over the appropriate records can be used to obtain the costs of individual types of parts.

Or one can sort the file on the "Unit Cost" field to see which parts are contributing the most costs to the machine. Perhaps substitutions can be made to reduce manufacturing expenditures?

Are you going to build some prototypes? A list arranged according to the stock numbers used in your plant can, possibly, speed up the parts-pulling procedure by stockroom personnel.

What? The parts are not in stock and they must be ordered from the vendors? No problem, your purchasing agents will think you are the world's greatest designer if you do the following:

Sort the data file according to vendor name. List the file while suppressing all the fields except the part description, quantity, vendor name, contact, and phone number. Give the list to purchasing. It will be ordered so that all the parts from each vendor are grouped together. (Warning, once you do this for a purchasing agent, be advised that they may become addicted to the service. Make sure your computer is reliable, for failing to deliver in this fashion in the future can bring loud complaints!)

## APPOINTMENT ORGANIZER

Are you a doctor, dentist, attorney, or businessperson who maintains an appointment schedule? If so, you may want to use your data base management program to help keep track of all your meetings. Here is one way to go about it:

Format a file so that records contain the fields illustrated here:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  LAST NAME                    14 A
 2:  FIRST NAME                   1Ø A
```

```
3:  DATE                          8 N
4:  TIME                          4 N
5:  PURPOSE                      40 A
```

Note that in this application, two fields are assigned to an individual's name. This was done because it may be desirable to sometimes organize the file alphabetically by last name. (The field called "First Name" can also hold a middle initial and/or title, such as "Jr." or "III.")

Note that the "Date" field is numeric in type. The convention mentioned in an earlier application, where dates are stored in the arrangement year-month-day, will be used. All four digits of the year are assumed to be used in the example. Thus, July 5, 1982, is entered as 19820705. Some may want to shorten this field to six digits, using only the last two digits of the year. In many situations it is likely that the year could be eliminated altogether. Thus, only four spaces—for the month and day—would be required.

The "Time" field is allotted four numeric digits. It is assumed that international "24 hour" time will be used in this example. This has advantages if you want to be able to chronologically order appointments occurring on the same date.

If a user wanted to use 12 hour notation, this field might be formatted as alphanumeric in type and assigned a length of six characters. Then the time could be prefixed by the notation "AM" or "PM" as appropriate. If this were done, you would want to be sure to always use four digits for the time in order to allow proper alphanumeric sorting of the field. Thus, 1:30 in the afternoon would have to be inputted as: PM0130.

The last field, labeled "Purpose," would be used for noting the intent of the meeting.

You might want to provide room for one or two more fields in the records. These fields could be used for storing notes about what was actually accomplished at a meeting.

### Line 'Em Up!

Once you have the record format determined, you can start lining up all those appointments by entering the appropriate information. Here are what a few entries might look like in a typical application:

```
LAST NAME: SMITH
FIRST NAME: JOHN P.
DATE: 19820705
TIME: 915
PURPOSE: ADVERTISING CONTRACT

LAST NAME: DOE
FIRST NAME: JANE
```

```
DATE: 19820705
TIME: 1000
PURPOSE: PRINTING SCHEDULE

LAST NAME: JONES
FIRST NAME: SAMUEL T.
DATE: 19820705
TIME: 1430
PURPOSE: DISCUSS NEW PRODUCTS
```

Since this type of application is likely to be an ongoing affair, you will want to save a copy or two of your current data file on a diskette. If your appointment schedule is vital to the conduct of your business or personal affairs, then it is a good idea to keep the file backed-up on several diskettes. One way to do this is to write the file to alternate diskettes on alternate days. That way, if you ever lose a diskette, you will only be out a day's worth of appointments.

There are a variety of ways in which you can use the power of your data base management program to help you keep track of your appointments.

For instance, once you have inputted your initial batch of future meetings, you might do the following:

Sort the file according to the contents of the "Date" field. Then list the contents of the file so that you see the grouping of your appointments by date. Now further arrange the file chronologically by time of day. You could do this by sorting the groups of identically dated records according to the contents of their "Time" fields. To do this, you would specify the range of record numbers over which each sort operation (on identically dated records) was to be performed.

The result would be a file arranged by date and time of appointment. A printout would provide you with a "master" list of future appointments.

Alternately, each day you could use the FIND option to produce a list of appointments for that day (by searching for occurrences of the current date in the "Date" field).

Perhaps you would rather have a list of meetings arranged according to the last names of the parties or by subject matter. No problem. Sort the file on the appropriate field and list it out!

### Practical Aspects

Frankly, this type of application is best approached from the "batch mode" point of view. It is highly unlikely that you will want to keep your computer tied up all day serving as an appointment scheduler. You are better off keeping a printed copy of the file (or portion thereof, such as the current day's appointments) handy. You can then check appointments off as they occur or make a note directly on the list to insert future

appointments. Then, at the end of the day, you can call up your appointments data base from a diskette and update it.

Trying to run this application in "real time" by constantly referring to the computer is seldom practical. While you might be tempted, for instance, to keep the appointment data base "on line," in most instances it simply won't save time. Unless the data base is relatively small (say, 3Ø to 4Ø entries or less), the time it takes to have the computer search for a particular entry can become impractical. After all, it is not likely that you would want to keep a potential customer on the phone more than 3Ø or 4Ø seconds while you tried to find a suitable time slot in which to schedule a meeting. You are better off having an up-to-date physical listing that you can refer to immediately. Then revise that list on a regular basis as you add or modify appointments.

## SELF-TUTORING

This is a simple, yet highly effective, application of the data base management program. Despite its simplicity and obviousness, many people are surprised to discover its use in the "educational" arena.

The whole idea here is to set up a file so that the program will serve as a computerized memory-drill aid. All you have to do is format records to contain a single field. This field will serve two purposes; which one depends on whether the record number is odd or even. You might format the record and assign the field name as:

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  ? / ANSWER                    4Ø A
```

You then build up the data file by alternating questions and answers in the records comprising the file. That is, record number 1 would contain a question. Record number 2, the answer to the question in record number 1. Record number 3 would contain another question, record number 4 the answer to that, and so forth.

The questions can relate to virtually any subject matter. But, it is generally best to structure the information as one might do for a vocabulary drill. Here is how a few questions and answers might appear in a hypothetical file (with each record preceded by its record number to highlight the odd/even arrangement of questions/answers).

```
( 1 )
? / ANSWER: CAPITAL OF CALIFORNIA?

( 2 )
? / ANSWER: SACRAMENTO
```

```
( 3 )
? / ANSWER: SQUARE ROOT OF 9?

( 4 )
? / ANSWER: 3

( 5 )
? / ANSWER: OPPOSITE OF OLD?

( 6 )
? / ANSWER: NEW

( 7 )
? / ANSWER: NUMBER OF CUPS IN A QUART?

( 8 )
? / ANSWER: 4
```

Since there is only one field in a record, you can put quite a few records in a file. (Particularly if you reduce the field length down to, say, 2Ø characters. If you have sufficient memory, it is possible to store up to 499 question/answer pairs in a file!) Once you have created the "drill" file, you can immediately put the "self-tutor" to work.

All you have to do is place the program in the LIST mode. Select the option that outputs to the video screen. Begin the drill with any odd-numbered record. (You can elect to have record numbers printed if you want to keep track of where you are in the file. Or, if you find them distracting, select the option that suppresses the display of the record numbers.)

When the odd-numbered record is displayed on the screen, it will present a question. Mentally answer the question. When you are ready to confirm your answer, press a key. Bingo! The next record that displays will give the answer to the previous question. Were you right or wrong? You have immediate feedback. Make a note of any questions you miss.

To bring up the next question, press a key. You are on your way. Drill as fast and as long as you want. The session terminates when you have gone through the file. Or, you can end the session by striking the RETURN key.

The system is neat and really works. Furthermore, if you have a large file covering a lot of facts, you can select the particular sections you want to cover at any session. Just tell the LIST option where you want to start within the file. (Keep a summary, of course, of the record numbers you use for questions/answers on a particular subject.)

The method is particularly suitable for vocabulary drilling when learning a new language. And, it is adaptable to just about any subject matter that requires the memorization of facts and figures.

Study hard and you too may grow up to be President.

## CHECKING ACCOUNT

Ever wish you had an easy way to reexamine all your checking transactions? Easier, that is, than simply scanning your scribbled-in checking account ledger or checking stubs.

Well, now you do! You can use your data base program to reorganize your checking account activity. Why, it can even help you reconcile your monthly banking statements!

Here is one way to approach the matter. (You know by now that the purpose of these application examples is just to shed some light on the subject. You add your own little touches to customize aspects to meet your own particular requirements.)

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

 1:  D/W                          1 A
 2:  AMOUNT                       9 N
 3:  DATE                         8 N
 4:  REF. NR.                     4 N
 5:  PAYEE                       20 A
 6:  PURPOSE                     20 A
```

The first field will be used to indicate whether the rest of the entries in a record are for *D*eposits or *W*ithdrawals. The use of this little, single-character field, makes it easy to sort the file into those two major groups.

The second field will store the dollar amount of deposits and withdrawals. To make statement reconciliation easier (that means have the computer do it!), entries in this field may be positive (deposits) or negative (withdrawals, i.e., checks).

The third field stores the date in the format: year-month-day. March 1, 1982, would thus be entered as: 19820301. The use of this format in a numeric field facilitates arranging the file in chronological order using the SORT option, if desired.

The next field will be used for recording the check numbers. It can also be used for numbering deposits.

Finally, there are fields for recording the payee and reason for the payment.

### Check It Out

Here are a few typical entries:

```
D/W: D
AMOUNT: 500
DATE: 19820301
REF. NR.: 1001
PAYEE:
PURPOSE:
```

```
D/W: W
AMOUNT: -102.12
DATE: 19820304
REF. NR.: 101
PAYEE: STATE NATIONAL BANK
PURPOSE: CAR PAYMENT # 17

D/W: W
AMOUNT: -58.29
DATE: 19820307
REF. NR.: 102
PAYEE: N. E. UTILITIES
PURPOSE: ELECTRIC SERVICE
```

The first entry illustrated is for a deposit to the checking account. Note that the "Amount" field contains a positive value. The "Reference Number" (REF. NR.) field contains 1001. The convention in this example being that deposits are numbered above 1000 to distinguish them from check numbers that will be less than that value. The last two fields are simply not used for deposit entries.

The second and third records indicate withdrawals by check. Note, particularly, that the contents of the "Amount" fields are negative in value (preceded by a minus sign) to indicate withdrawals.

### Putting It to Work

Want to know what your present account balance should be? Just do a TALLY operation over the entire file on the contents of the "Amount" field. A positive result means you are ahead of the game and have an available balance. A negative answer means you are overdrawn. Oh-oh.

Having difficulty reconciling a statement from the bank? Try sorting the file on the first field. This will split it into two major sections: deposits and withdrawals. If you are assigning reference numbers to deposits, then you might want to order that section of the file according to those numbers. Just sort the appropriate range of record numbers by the contents of the fourth field (REF. NR.). If you are not using reference numbers for deposits, then perhaps a sort on the "Date" field (over just those record numbers representing deposits) will be helpful.

Now finish organizing the withdrawals. Sort the appropriate range of records by the contents of the "Reference Number" field. Then use the LIST option to print out just the "Amount" and "Reference Number" (and possibly the "Date") fields for the entire file. Remember, you do this by *suppressing* all of the *unwanted* fields. In just a few moments you will have a neatly organized list of deposits and withdrawals. You can compare these against your bank statement and returned checks. You can also obtain separate tallies over just the deposit records and just the withdrawal records, to get subtotals for each category. With this kind of assistance, you should be able to find those mistakes made by that big old bank in no time at all.

You say it is the end of the year and you want to see how much you paid out to individual payees? Fine. First you sort by the contents of the "Payee" (or perhaps, "Purpose") field. Once the file is grouped according to payee, then you can do a tally over each subgroup. This will give you the amount paid to each payee over the entire time spanned by the file. (Watch out, this exercise can sometimes be shocking. It is no fun learning that you have been paying $45Ø.ØØ a year just to be able to see in the dark! Is it possible that candles would actually cost less?)

By the way, did you know that the FIND operation can look for the occurrence of a character string *anywhere* within a field? Yes, it can. That means, for instance, that it might be interesting to do something like this: Select the FIND option and tell it to print out all the records that contain the character string "CAR" in the "Purpose" field. It will give you a list of all payments that you indicated as having any reference to cars: repairs, registration, and inspection fees, etc.

A little thought with that fertile imagination of yours and I am sure you will see lots of other possibilities.

## INVESTMENT PORTFOLIO

One of the aggravations of playing the stock market (besides buying at the top and selling at the bottom) is keeping track of all of your buy/sell transactions. But accurate records are a must. Not only do they tell *you* the score, but they are required in order that you can correctly tell Uncle Sam *his* score.

A data base file formatted along the following lines can be a big help in maintaining accurate records and keeping yourself up to date on the overall performance of a portfolio.

```
FORMAT BY FIELD #, NAME, SIZE, TYPE:

SYMBOL                    6 A
COMP. NAME               2Ø A
# SHARES                  4 N
BUY PRICE                 6 N
BROKER FEE                4 N
BUY DATE                  8 N
SELL PRICE                6 N
TRANS. FEE                4 N
SELL DATE                 8 N
NOTES                    4Ø A
```

The field assignments for this application are pretty straightforward. The first two are reserved for the ticker-tape symbol and name of the company.

The third field is numeric, as are all of the remaining ones except the last. It is used to hold the number of shares involved in the transaction.

Then there are fields for the buy price and broker fee. In this example it is assumed that the *total* price for the lot is entered as well as the *total* broker's fee. Using totals here allows the data base management program's TALLY operation to perform more valuable functions than if the per-share price was recorded. (Of course, *you* could always add another field to hold this information!) It is also assumed that the price is entered to the nearest whole dollar. (If you want to keep track of prices right down to the last cent, then you might want to allow a few more digits in the price-related fields.)

The sixth field stores the date on which the assets were purchased. Note that eight digits are reserved for the date. The format suggested for this application is: year-month-day, such as 1982Ø2Ø5 for February 5, 1982. This format allows the SORT option to arrange the file in chronological order if the "Date" field is selected as the key.

This is followed by fields for the *total* price at which the investment was sold and any selling-related fees.

The next field stores the date on which the assets were no longer owned by you.

Finally, there is an alphanumeric field for keeping miscellaneous notes about the transactions.

### Update as You Go Along

In this type of application, you would typically keep your file on a diskette. This file (you might name it "STOCKS") would be loaded into memory whenever you wanted to update it. After updating, you would save the revised edition back on a diskette.

A typical entry might appear as shown here:

```
SYMBOL: ATT
COMP. NAME: AMERICAN TEL & TEL
# SHARES: 1ØØ
BUY PRICE: 5725
BROKER FEE: 125
BUY DATE: 19820205
SELL PRICE: 6ØØØ
TRANS. FEE: 35
SELL DATE: 19820226
NOTES: SHORT TERM GAIN
```

Of course, until you actually disposed of an investment, fields 7, 8, and 9 (and possibly 1Ø) would not be used.

### Analyze When Ready

You can analyze your portfolio anytime you want. First, of course, you can always get a neat listing of all or part of your transactions to date. This listing can be arranged according to buy or sell date (chrono-

logically) or by the stock's symbol or corporate name. Just sort on the desired field before obtaining the listing. Remember, too, that you can suppress unwanted information from the display or printed list.

But there are much more powerful options than just having a neatly arranged list. You can tally on the "Buy Price" and then "Sell Price" fields. Depending on the condition of your file and the range of records over which you perform the summation, you can obtain several vital figures.

If your file is arranged chronologically by sell date, then you will be able to separate those stocks that have been "turned-over" from those that you currently hold. You can then tally over the appropriate range of records to determine your total current investment holdings. Or you can tally the buy price and then the sell price over the range of records that have been "turned-over." Subtracting the sell total from the buy total gives you a gross profit. (You are doing this for profit, aren't you?) You can obtain your net by factoring in broker's and transaction fees. (Those, too, can be obtained using an appropriately bounded TALLY operation.)

If, as shown in the example, you make entries such as "loss" or "gain" in the "Notes" field, you can make additional use of the FIND option. Thus, if you want to ascertain all your loss issues or study your gains, then you merely specify "loss" or "gain" in the search directive. The computer will dutifully bring up those transactions for your review.

### Now You Are on Your Own

The many examples that have been provided in this section are intended to give you *ideas.* You are now aware of many ways in which this highly versatile program can be applied. Don't be afraid to experiment. The only thing you have to lose is a little time—time that you will gain back through the more effective use of this powerful tool that such experimentation invariably brings!

May all your applications of the program be profitable.

# A TECHNICAL OVERVIEW

The remaining material in this book is provided to satisfy the curiosity of advanced users. It provides technical information on the structure of the program and its implementation. A thorough understanding of the material in this section will be helpful to those users who might want to expand or modify the capabilities of the program.

It should be emphasized that this section is intended primarily for those readers who have a knowledge of programming. *It is not necessary to read this section in order to utilize the program.*

## Program Design Philosophy

Any programmer who creates programs must develop or select certain guidelines in the design stages. These criteria influence various decisions. Tradeoffs must be made between various options.

Some of the key considerations during the development of this program centered around these factors: *simplicity*, so that the basic operation of the program might be understood by others; *ease of use* by unskilled operators; *modularity* so that advanced users could consider and implement design changes to meet their own special requirements; *compactness* so that sufficient memory would be left available for storage of a meaningful data base; and *I/O integrity*, so that inadvertent operator directives would not result in the unintentional destruction of the data base or disruption of the program.

The factors mentioned in the preceding paragraph, for instance, were given more importance than such parameters as speed of program execution, efficiency of operation (from an algorithmic viewpoint), or fancy screen displays.

Of course, every programmer tends to emphasize different parameters. The choices are often a matter of personal taste. Consequently, some readers may disagree with the design criteria I selected. That is fine. Everyone is entitled to their opinion. And with the

information provided here, everyone with sufficient programming knowledge, whether they agree or disagree with any or all of the design aspects, will be free to proceed to modify and adapt the program as they see fit.

### How the Data Is Organized

Each *record* entered by the user is stored in a string array labeled B$. This array is dimensioned as B$(999). Thus, theoretically, up to 1ØØØ records can be stored. However, due to I/O limitations and other considerations, the user is restricted to using records that are referenced by numbers in the range of 1 through 999. Since array elements are numbered beginning with Ø, the physical array elements are always one less in value than that referenced by the user. Hence, when the user specifies manipulation of record number one, the program will actually deal with array element number zero. Because of this, array element number 999 is never actually used to store a record (as it would be the one-thousandth). Thus, this element effectively serves as a "guard" element. There are times, when performing selected data base manipulating operations, that having such an extra element available can be useful. (Can you think of any?)

Each element of the array used for storing records is limited by I/O bounds (placed in the program) to a maximum of 236 characters. The theoretical limit for a string array element (imposed by Applesoft) is 255 characters. I stayed shy of this limit to: (1) provide a few "guard" characters during operations such as CHANGE when it is necessary to split records in order to insert changes, and (2) leave room for advanced disk operations that might require the use of additional "guard" characters.

### Control Arrays

Several other arrays are used to supplement and control the manipulation of information stored within records. One is referred to as the Field Name Array and has the variable designation A$(Ø) through A$(39). Thus, there can be up to 4Ø named fields in a record.

A numeric array (versus a string array) having elements A(Ø) through A(83) serves several other purposes. The first 4Ø elements, A(Ø) through A(39), are used to store the number of characters assigned to each field. The next 4Ø elements, A(4Ø) through A(79), are used to indicate whether the field is alphanumeric or numeric. If the associated field is alphanumeric, then these elements contain the value zero. If the associated field is numeric, then the corresponding element contains the value one.

Element A(8Ø) is used to store the total number of fields a user assigns to a record format.

Element A(81) holds the current file record count or end of file marker. When used as a counter, its value is equal to the actual number of records present in the file. When used as a pointer, it points to the next array element in B$( ) where a record may be *appended* to the file.

Element A(82) stores the maximum number of records a file is permitted to hold. (This value is a function of the record format and the amount of user memory available in a system.)

Element A(83) stores the maximum number of characters allotted to a record. It is obtained by summing the number of characters permitted in each field of a record format, that is, the values held in elements A(Ø) through A(39) for whatever number of fields have been assigned.

Still another array is used during selected I/O operations. This array consists of elements B(Ø) through B(39). During, for instance, the LIST operation, the elements corresponding to assigned fields contain the value zero if the associated field is to be displayed or printed. They contain a one if the output is to be suppressed.

### Storing Files

The current status of any data base being manipulated by the program is easily stored for later recall. All that is necessary is that the A( ), A$( ), and B$( ) array elements be saved. This is what is actually accomplished when the user elects to save a data base on a diskette.

Restoring a file from diskette to memory is just a reverse of the process indicated in the previous paragraph. The array elements are loaded into memory. They contain all the information necessary for the program to ascertain the format of the file along with the information contained in the current data base.

### Major Routines

The following discussion will outline the operation of each major portion of the data base management program. For a detailed analysis, refer to the actual program listing in Chap. 3 and the line-by-line commentary of the listing in Chap. 6.

First, it may be noted that the program is highly modular. Each particular function or operation is assigned a block of line numbers for easy reference. Here is a summary of the major routines arranged according to their starting line numbers:

```
1000  -  RECORD FORMATTING ROUTINE
2000  -  APPEND ROUTINE
3000  -  INSERT ROUTINE
4000  -  CHANGE ROUTINE
5000  -  DELETE ROUTINE
6000  -  LIST ROUTINE
```

```
7000  -  FIND ROUTINE
8000  -  SORT ROUTINE
9000  -  TALLY ROUTINE
10000 -  ERASE FILE ROUTINE
11000 -  DISK CATALOG ROUTINE
12000 -  PROGRAM EXIT ROUTINE
20000 -  READ DATA BASE FILE FROM DISKETTE
30000 -  SAVE DATA BASE FILE ON A DISKETTE
40000 -  PRIMARY MENU ROUTINE
41000 -  SECONDARY MENU ROUTINE
50000 -  ALPHANUMERIC INPUT SUBROUTINE
51000 -  NUMERIC INPUT SUBROUTINE
59500 -  ERROR MESSAGES
59800 -  MISCELLANEOUS UTILITY SUBROUTINES
```

**The Record Formatting Routine**—This routine first checks to make sure that a file is not already present in memory. If one is present, the routine immediately exits to the Primary Menu.

To format a new file, the routine accepts operator inputs specifying field names, length, and type. Field names are stored in the appropriate elements of the Field Name Array A$( ). Field lengths and type designations are placed in the appropriate elements of A( ). Inputs are tested to verify that field length and record length restrictions are not exceeded. This routine loops until a maximum of 40 fields has been defined, the allowable record length is reached, or the user indicates that the necessary fields have been defined. The routine normally exits to the Secondary Menu.

**The APPEND Routine**—This routine first checks to make sure that the file is not already filled. Provided that it is not, it accepts input to the next available record position on a field-by-field basis. The routine displays prompts that include the record number, field name, maximum allowable length, and type. Inputs are tested and constrained to meet the requirements of the record format. The routine loops until the file is full or the user indicates that the appending session may be terminated. The routine normally exits to the Secondary Menu.

**The INSERT Routine**—Upon entry, the routine checks to see that the file is not already filled. Provided that it is not, the routine queries the user for the file position at which new records are to be inserted. The user-specification is tested for validity. Upon validating an insertion position, the routine "expands" the current file to make room for the inserted record. This is accomplished by moving all of the records above the insertion position (in the B$ array) up one element. The routine then accepts user inputs on a field-by-field basis, just as is done for the APPEND operation. Inputs are tested and constrained to conform with the current record format. When a record has been inputted, the routine queries the user as to whether another record is to be inserted at the current position. If so, the routine repeats the procedure. This routine normally exits to the Secondary Menu.

**The CHANGE Routine**—Upon entry, this routine ascertains that the file is not empty. Provided that it is not, the routine prompts the user to input the field numbers indicating which fields within a record are to be changed. The routine then prompts for the number of the first record that is to be modified. It then accepts user inputs of new data for those fields that the user previously indicated would be altered. Prompts are shown for each field. User inputs are tested and constrained to conform to the record format. When all the relevant fields have been inputted, the operator is asked if the next record in the file is to be changed. If so, the process is repeated. The routine normally exits to the Secondary Menu.

**The DELETE Routine**—The routine first checks to make sure the file is not empty. The user is then prompted for the range of records to be removed. This specification is checked for validity. If valid, the routine moves all records higher in number than the upper bound specified by the user, down to the lower bound specified by the user. This is accomplished by directly manipulating the contents of the B$( ) array. Record positions vacated (at the end of the file) by this operation are then nulled. The file's record counter, A(81), is then updated to reflect the file's current status. That is, the number of records removed is subtracted from the original value in A(81). The revised record count is displayed for the user at the conclusion of the deletion process. The routine normally exits to the Secondary Menu.

**The LIST Routine**—The routine initially checks that the file is not empty. Provided that it is not, it reminds the user of the highest numbered record in the file. It then prompts for the range of records to be outputted. Next, the routine asks for the numbers of any fields that are to be suppressed. Inputs are tested for validity. The user is given the option of having record numbers suppressed. The user also dictates whether output will be to the system's screen or an external printer. At the beginning of the listing operation, the names of the fields, their maximum length, and type are outputted as a header. The records requested are then outputted to the appropriate device. This routine normally exits to the Secondary Menu.

**The FIND Routine**—The routine tests to see that the file is not empty. If not, it reminds the user of the highest numbered record in the file. It then prompts for the range of records to be searched. Next, the routine asks for the numbers of any fields that are to be suppressed. Inputs are checked for validity. The user can specify whether record numbers are to be displayed and selects output to the screen or printer. Then the user is prompted to give the name of the field that is to be searched within the records. If an invalid name is inputted, the routine displays a list of the field names assigned to records in the file. Finally, the user inputs the character string that the routine is to look for within the specified field. The routine sets up a pointer to access the field that is

to be searched within each record. A header is displayed showing the range of records being scanned and the search criteria, then the names of the fields, their maximum length, and type. The search string is then matched against all possible positions within the field being examined. If a match is found anywhere within the field, then the unsuppressed fields of the record are outputted to the appropriate device. The process continues until the specified range of records has been examined. However, if the output is being directed to the video screen, the process can be terminated earlier by an appropriate user response. The routine normally exits to the Secondary Menu.

**The SORT Routine**—The routine initially checks that the file is not empty. Then, after displaying the highest numbered record in the file, it queries the user for the starting and ending numbers (range) of records that are to be sorted. The range specified is tested for validity, including the requirement that it span at least two records. The routine prompts for the name of the field that is to be used as the sort key. If a valid name is not specified, the current field names are reviewed for the user. A pointer is set up to access the key field. The portion of the file specified is then arranged in alphabetical or numerical (if the key field is numeric) order. The method used is commonly known as the Shell Sort. Note that when a test of the key field indicates that a change in position needs to occur, then the entire contents of the records involved are swapped within the file. Since a sorting operation can take a number of minutes for a large file, a "sorting" message is displayed during the process. This routine normally exits to the Secondary Menu.

**The TALLY Routine**—The routine checks that the file is not empty. It then advises the user of the highest numbered record in the file. Next it prompts for the range of record numbers that are to be summed. The range specified is checked for validity. It then asks for the name of the field that is to be tallied. If an invalid name is given, the routine displays the current list of field names. If the field named is not numeric in type, the routine notifies the user. The procedure is then terminated by exiting to the Secondary Menu. If the field is numeric, then the contents of that field are summed over the range of records specified. The results of the summing operation are then displayed to the operator. Upon cue from the user, the routine exits to the Secondary Menu.

**The Erase File Routine**—This routine first verifies that the user does indeed want to erase the current data base file in memory. If so, it clears all variables. This effectively erases any data file that was in memory. The routine then exits to the Primary Menu.

**The Disk Catalog Routine**—After establishing an error trap to capture any I/O errors, this routine uses the standard DOS directive to display a diskette's catalog. An I/O error condition causes the routine to terminate (with an appropriate message to the user) and exit to the

Primary Menu. Otherwise, the routine performs a normal exit, at the conclusion of the cataloging process, to the Primary Menu.

**The Program Exit Routine**—This routine advises the user that leaving the program will destroy any data base currently in memory. It then prompts the user to confirm the directive. If it is affirmed, the program is terminated by exiting to the system monitor. Otherwise, the routine returns to the Primary Menu.

**The Read Data Base File Routine**—This routine initially checks to see that there is not already a file in memory. If there is, the routine immediately exits to the Erase File routine. Otherwise, the user is prompted for the name of the file that is to be inputted from a diskette. An I/O error trap is established. DOS commands are then used to open the specified file and read in data in the following order: the A( ) array (field characteristics), the A$( ) array (field names), and the B$( ) array (actual data records). Should there be an I/O error during this process, the routine is exited via an error routine. Otherwise, the routine normally exits to the Primary Menu.

**The Save Data Base File Routine**—The routine first checks to ascertain that a data base file, containing some data, is present in memory. If not, the routine is immediately terminated by exiting back to the Primary Menu. Provided that a data file is present, the routine prompts the user for a file name under which to store the data file on a diskette. The routine establishes an error trap for use during disk I/O operations. It then uses DOS directives to create a file with the user-specified name on a diskette. The method used will overwrite any previously existing file with the same name. Once the file has been created, the data file in memory is written to the diskette in the following order: the A( ) array (field and file parameters), the A$( ) array (field names), and the B$( ) array (data records). Should an I/O error occur during the process, then the routine exits via an error routine. Otherwise, the routine concludes by exiting back to the Primary Menu.

**The Primary Menu Routine**—This routine displays the primary program options available to the user on the screen in menu format. It then "captures" the keyboard and waits for a valid menu selection. Invalid selections are ignored. When a valid menu option key is pressed, program control passes to the appropriate routine.

**The Secondary Menu Routine**—This routine displays the secondary program options available to the user on the screen in menu format. It then scans the keyboard, waiting for a valid menu selection. Invalid selections are ignored. Pressing a valid menu option key results in the program passing control to the appropriate routine.

**The Alphanumeric Input Subroutine**—This routine establishes an input buffer and accepts keyboard inputs to this buffer. The routine

deletes the previous character if the backarrow character is detected, unless the buffer is empty. The routine also limits the number of characters inputted to the buffer. The routine exits to the calling routine when the code for the RETURN key is detected, unless the buffer is empty. The RETURN key is ignored when the input buffer is empty.

**The Numeric Input Subroutine**—This routine calls upon the Alphanumeric Input subroutine to fetch a string of characters from the keyboard. It then checks this string to ascertain that it contains a valid numeric input. If an "integer" flag is set, it will also verify that the value received is strictly integer. If the string is found invalid, the content of the input buffer is erased, an appropriate error message is displayed, and the input process is repeated. Receipt of a valid numerical entry causes the routine to exit to the calling routine.

**Error Messages**—Error messages used by more than one routine are grouped together in this part of the program for reference purposes. These messages are placed into a string variable as required by the calling routine. Program control then passes back to the calling routine.

**Miscellaneous Utility Subroutines**—Commonly used "utility" subroutines that are used to control and format the screen display are grouped at the end of the program.

### Notes about Line Numbering

Line numbers are separated by increments of at least 10. Where possible, logical breaks within routines are indicated by resuming numbering at the next hundreds level. Subroutines are grouped at the high end of each section.

Thus, for example, the Record Formatting routine has subdivisions that begin at lines 1000, 1100, 1200, and 1400. It also has a subroutine that begins at line 1900.

This line numbering convention provides plenty of room for users to insert additional instructions within each section. Such insertions can generally be accomplished without interrupting the overall structure of the rest of the program.

There are also plenty of areas in which to place entire new sections if desired. For instance, blocks of line numbers for new routines could be assigned in the range 13000 through 19999.

Refer to the program listing in Chap. 3 and the line-by-line program commentary in Chap. 6 for more detailed information on the program's structure and operation.

# 6 PROGRAM COMMENTARY

This chapter is devoted to providing the kind of documentation that programmers (and potential programmers) generally find useful. It consists of three main sections: (1) a detailed line-by-line commentary of the BASIC source listing, (2) a table of variables usage, and (3) a table of line number cross-references.

The line-by-line commentary refers to the individual program lines contained in the BASIC source listing. (That listing is provided at the end of Chap. 3.) It is hoped that reference to this commentary along with the actual listing will provide the interested, qualified programmer with about all there is to know about the operation and "why-for" of the data base management program. Chances are good that novice programmers, too, will be able to learn something from this detailed explanation.

The table of variables usage is prefixed by a brief summary of the major purpose(s) assigned to each variable name. The table itself is a valuable reference if you are planning to make any alterations to the program. It enables you to ascertain each and every line in which the variable is used. Thus, you can check that any changes you propose (for the usage of those variables) will not unduly impact the operation of another section of the program.

Finally, the table of line number cross-references further assists the adventurous program modifier. It lists each line number that contains a reference to other line number(s), such as through GOTO, GOSUB, and THEN statements. If, for instance, you decide you want to remove a block of code from the program or change the operation of a subroutine, you check this table. By doing so you can ascertain whether any other statement lines refer to the section you are eliminating or changing. If so, check each reference to make sure your alterations will not upset the apple cart.

With all this information at your fingertips, you will be in a good position, if you are so inclined, to: (1) learn just about all there is to know

about the actual operation of the program, and/or (2) proceed (perhaps with the help of the next chapter) to modify and enhance the program to suit your own special needs and preferences!

## Line-by-Line Commentary

**1** Jump to Primary Menu at program startup.

**1000** Check to see that file has not already been defined.

**1010** If file already exists, then clear screen. Notify user of the condition. Give advice. Set delay value. Call delay routine to provide user time to read message.

**1020** Go back to the Primary Menu.

**1030** Entry point for defining file. Initialize all variables. Call subroutine to DIMension arrays.

**1100** Clear screen.

**1110** Prompt user for name of a field.

**1120** Limit name length to 10 characters. Accept user's alphanumeric input.

**1130** Stuff field name into Field Name Array.

**1200** Provide some blank lines on the display.

**1210** Prompt user for length of field. Advise user of the maximum permitted.

**1220** Limit input to 2 digits. Specify integer only. Accept numeric input.

**1230** Check length specified. If not valid, set up error message. Call error display routine. Erase last input from display. Loop back to try again.

**1240** See if new field length would result in excessive record length. If so, set up error message. Call display routine. Erase last input. Loop back to try again.

**1250** Stuff valid length into appropriate element of Length Array.

**1260** Prompt user to classify field as (A)lphanumeric or (N)umeric.

**1270** Input limited to one character. Accept alphanumeric input.

**1280** If field designated alphanumeric, then stuff a '0' into appropriate Field Status Array element. Skip ahead.

**1290** If operator does not respond appropriately, then set up an error message. Call error display. Erase input. Loop back to try again.

**1300** Stuff a '1' into the Field Status Array for a numeric field.

**1310** Update record character count. If maximum permitted length reached, then force a halt to field definitions.

**1320** Also force a halt if have reached maximum number of fields allowed (40).

**1330** If program reaches here, then can clear the display.

**1340** Ask user if another field is to be defined.

**1350** Limit response to one character. Accept alphanumeric input.

**1360** If user is continuing, then increment field counter. Loop back to define next field.

**1370** If operator does not respond appropriately, then set up error message. Display message. Erase input from screen. Loop back to try again.

**1400** If user finished defining fields, then clear the screen.

**1410** Acknowledge conclusion of field definitions. Provide a few blank lines on the display.

**1420** Set element A(80) of Field Length Array to indicate the actual number of fields in a record. Initialize element A(81) of this array to serve as a counter of the number of records stored in the file.

**1430** Now size up memory—leave some room for variables and possible future modifications by the user. Save the number of characters per record in element A(83). Determine the number of records that can fit in available memory. Limit the maximum number of records to 999 to avoid DIMensioning problems.

**1440** Notify users as to how many records may be stored. Save the maximum allowable number of records in element A(82).

**1450** Set delay value. Provide delay so user can read display. Set File Defined flag. Go to Secondary Menu.

**1900** Set File Defined flag. DIMension arrays used by data base. Return to caller.

**2000** APPEND ROUTINE. Call subroutine to see if file is full. If so, exit right back to the Secondary Menu.

**2100** Initialize record Character Position counter and the Field counter. Clear current record to null condition.

**2200** Clear display. Convert current record number to string format for neater display. Throw function notice on the screen. Provide a few blank lines to delineate title.

**2210** Call subroutine to display Append status and accept input for a field.

**2220** Append the field data to the current record.

**2230** Skip ahead if have accepted as many fields as have been specified for a record.

**2240** Otherwise, update the Character Position counter. Then increment the Field counter. Loop back to get the data for the next field.

**2250** Update the Records counter.

**2260** Provide separation on the display. Ask if user wants to enter another record.

**2270** Limit response to a single character. Accept alphanumeric input.

**2280** If user appending another record, then jump back to the start of this routine.

**2290** Otherwise, exit to the Secondary Menu.

**2800** Subroutine to accept data in fields. Display current record number and show the maximum number of characters permitted in the current field.

**2810** Check to see if current field is alphanumeric. If so, display ALPHA and skip next line.

**2820** When field is numeric, tell the user.

**2830** Now display the field number and the user-defined name of the field.

## Line-by-Line Commentary (continued)

**284Ø** Display a line of dashes to separate the header from the user's input. Provide a few blank lines for additional separation.

**285Ø** Limit the user's input to the number of characters defined for the current field. If field is numeric, then clear the Integer flag to allow floating point values. Accept numeric input. Skip next line if field is numeric.

**286Ø** If field is alphanumeric, then accept alphanumeric input.

**287Ø** If input does not fill field, then fill rest of field with spaces.

**288Ø** Exit subroutine by returning to caller.

**29ØØ** Subroutine to determine whether file is full. Clear the Error flag. See if the current number of records in the file equals (or exceeds, for good measure) the number of records allowed in the file. If it does, then clear the display and throw up the 'File is Full' message. Set up the delay subroutine. Provide some time for the user to get the message.

**291Ø** Exit to the caller.

**3ØØØ** INSERT ROUTINE. Call subroutine to see if file is full. If so, exit right back to the Secondary Menu.

**3Ø1Ø** Call subroutine to see if file is empty. If Error flag is set upon return, then file is empty so clear the flag, provide a few blank lines, then tell the user to use the APPEND routine. Set delay value. Call delay routine so user has time to read message. Return to the Secondary Menu.

**31ØØ** Call routine to display function header. Call subroutine to identify last record in the file. Provide a couple of blank lines as a separator.

**311Ø** Query user for the position at which to start inserting record(s).

**312Ø** Limit response to maximum of 3 digits, specify integer values only. Accept numeric input. If user inputs zero (Ø), then abort this function and return to Secondary Menu.

**313Ø** If record specified is not within the range of those currently in the file, then set up an error message. Call the subroutine to display error message to the user. Erase the last input line. Loop back to try again.

**314Ø** Form a loop going from the highest numbered record in the file on down to the record specified by the operator. (Remember, the array element is one less.) Move all the records in this range 'up' one position to make room for the insertion.

**315Ø** Initialize Character Position counter and Field counter. Convert insertion position to a string value to enable a neater display. Null the Record array at the insertion position.

**316Ø** Display the function header. Provide a few blank lines as separator. Call subroutine to input data for the current field.

**317Ø** Add the field to the record being inserted. See if have processed all fields in the record. Skip next line if so.

**318Ø** Otherwise, update the Character Position counter and also increment the Field counter. Go back to get next field.

**3200** Update Records counter. Call subroutine to see if file is full. If Error flag is set here, then clear the flag and exit to the Secondary Menu as file is full.

**3210** Otherwise, provide couple of blank lines and query user about inserting another record.

**3220** Limit response to 1 character. Accept alphanumeric input. If response is not (Y)es, then exit to Secondary Menu.

**3230** If (Y)es, then advance Insertion pointer. Go back to insert another record.

**3900** Subroutine to display header for the INSERT operation.

**4000** CHANGE ROUTINE. Call subroutine to see if file is empty. If flag is set, clear flag, go back to the Secondary Menu.

**4010** Display function header. Clear Field Status elements (used to mark fields that will be changed).

**4020** Provide a couple of blank lines. Query user for the fields to be altered.

**4030** Call subroutine to identify fields that are to be altered.

**4040** Form loop to scan Field Status array elements. Check to see that at least one field is marked for change. If find any, then raise a flag.

**4050** Loop until array scanned. If flag set, then clear flag and skip over next line to start processing changes.

**4060** If there are no fields marked for change, then return to the Secondary Menu.

**4100** Display function header. Call subroutine to display the number of the last record in the file. Provide a couple of blank lines as a separator.

**4110** Query user for number of the first record to be altered.

**4120** Limit response to three digits, integer only. Accept numeric input. Abort to Secondary Menu if record number Ø is specified.

**4130** If record number specified is not valid, then set up an error message. Display message. Clear line just inputted. Go back to try for a valid record number.

**4140** Initialize Character Position counter. Convert record number to a string for a neater display.

**4150** If field not marked for change, then skip over "changes" section.

**4160** "Changes." Display function header. Provide a few blank lines after the header. Call subroutine to input data to a field.

**4170** Throw parenthesis around record being processed as "guard" characters.

**4180** Save everything in the record up to the field that is being modified. Append the new field data. Then append the rest of the original record.

**4190** Now strip off the "guard" characters.

**4200** See if have processed all fields in the record. If so, then skip next line.

**4210** Update Character Position counter. Increment the Field counter. Go back to process next field.

**4220** If have processed all records in the file, then exit to the Secondary Menu.

**4230** Otherwise, provide a couple of blank lines, then query the user as to whether the next record is to be altered.

## Line-by-Line Commentary (continued)

**424Ø** Limit response to a single character. Accept alphanumeric input. If response if not (Y)es, then exit to Secondary Menu.

**425Ø** Otherwise, advance Records counter and go back to do next record.

**49ØØ** Subroutine to display header for the CHANGE operation.

**5ØØØ** DELETE ROUTINE. Call subroutine to see if file is empty. If flag set, then clear flag and exit to Secondary Menu.

**51ØØ** Clear display. Display function header. Show number of the last record in the file.

**511Ø** Call subroutine to get range of records to be deleted. If flag is set, then clear the flag and abort operation by going back to the Secondary Menu.

**512Ø** Clear screen. Acknowledge records specified by user.

**513Ø** Ask user to verify range of records being deleted.

**514Ø** Limit response to 1 character. Accept alphanumeric input. If response not (Y)es, then go back to re-specify range.

**52ØØ** Calculate the number of records to be deleted. If the highest numbered record being deleted is less than the highest numbered record currently in the file, then form a loop. Set the loop counter for the number of records that need to be shifted in the file. Shift the records beyond those being deleted over the former records to take up the gap. Loop until all the remaining records beyond the gap have been moved.

**521Ø** Set a loop counter equal to the number of records being deleted. Decrement the Records counter. Null out the now unused Records Array element. Loop until finished. Display the number of records left in the file. Exit to the Secondary Menu.

**57ØØ** Subroutine to get range of records to be processed. Clear flag. Provide a few blank lines on the display. Query user for the first record number.

**571Ø** Accept user input. If user inputs 'Ø', then set the flag as operation is to be aborted. Return to caller.

**572Ø** If number specified is valid (within range of records currently in the file), then skip next line.

**573Ø** Otherwise, set up error message. Display message. Clear list line of input from the screen. Go back for a better input.

**574Ø** Save starting record number. Ask user for last record number to be processed.

**575Ø** Accept user input. If user inputs 'Ø', then set flag as operation is to be aborted. Return to caller.

**576Ø** If number specified for last record is less than first number given or more than highest record number in the file, then set up an error message. Display the message. Erase the last user input from the display. Go back to try for a new value.

**577Ø** Save the 'ending' record number. Exit to caller.

**579Ø** Subroutine to accept numeric input having a maximum of three digits. Only integer values will be accepted.

**5800** Subroutine to see if file is empty. Clear flag. See if Records counter is zero. If so, set error flag and clear the display. Notify user that file is empty. Set up a delay value. Call delay routine so user can read message.

**5810** Return to caller.

**5900** Subroutine to display number of the last record in a file. Show current Records counter value. Set delay value, give user time to read screen, then exit to caller.

**6000** LIST ROUTINE. See if file is empty. If so, clear flag and exit to Secondary Menu.

**6100** Display function header. Then display the number of the last record in the file.

**6110** Call subroutine to get range of records to process. If flag is set upon return, abort operation and exit to the Secondary Menu.

**6200** Call subroutine to determine which fields user wants to suppress during the listing operation.

**6210** Clear the screen. If Printer flag (C) is set, then issue command (in DOS format) to turn on external printer. Issue a few linefeeds.

**6220** Call the subroutine to display the listing format.

**6230** Form a loop over the range of records specified. (Remember that the array element is one less.)

**6240** Call subroutine to output a record.

**6250** Loop until have output records requested.

**6260** Provide some more linefeeds so user can tear off paper. Turn off the external printer. Go back to the Secondary Menu.

**6500** Subroutine to find out what fields user wants to suppress during a listing. Call subroutine to clear Field Status array elements. Then call subroutine to ask if user wants to suppress any fields.

**6510** Limit response to a single character. Accept alphanumeric input. Skip next line if response is not (Y)es.

**6520** Call routine to display operator directive. Call the subroutine that has user input the fields that are to be suppressed.

**6530** Call subroutine that asks if user wants to suppress the display of record numbers.

**6540** Limit response to one character. Accept alphanumeric input. If response is (Y)es, then set the Suppress flag (D) and skip the next line.

**6550** Clear the Suppress flag as record numbers are to be shown.

**6560** Call subroutine that asks whether user wants to use screen or printer.

**6570** Limit response to a single character. Accept alphanumeric input. If response is (P)rinter, then set the Printer flag and return to the caller.

**6580** If response is invalid (neither P nor S), then set up error message. Display the message. Clear the previous input from the display. Go back and try for a valid response.

**6590** If response is (S)creen, then clear the flag and return to the caller.

**6600** Subroutine to display record format. First display a title line.

**6610** Form a loop to scan the Field Status array elements. If the field is marked (1), then suppress output by jumping ahead.

## Line-by-Line Commentary (continued)

**662Ø** If field number is less than 1Ø (only has 1 digit), then move over one space to make neat display.

**663Ø** Now output the field number and user-defined field name. Use the length of the field name to calculate the number of spaces needed to keep the display neatly formatted. If the number of characters permitted in the field is less than 1Ø, add one more space to keep the display tidy.

**664Ø** Display the field character size. Provide a space. If the field was specified as numeric, display 'N' for (N)umeric, then skip next line.

**665Ø** Otherwise, display 'A' for (A)lphanumeric.

**666Ø** Loop to output the format for the next field. When have finished, issue a couple of blank lines and exit to the caller.

**67ØØ** Subroutine to output a record. Jump ahead if the Suppress flag is set.

**671Ø** Otherwise, output the record number enclosed in parenthesis.

**672Ø** Initialize the Character Position counter. Form a loop to check all fields in the record. Skip next line if Field Status element is set as this field is to be suppressed.

**673Ø** Output the contents of the field.

**674Ø** Update the Character Position counter. Loop back to output the next field. Add one linefeed between records.

**675Ø** If using printer, just return to the caller.

**676Ø** When outputting to the screen, provide a few blank lines. Give user the option of continuing listing or aborting.

**677Ø** Wait for a key depression. If it is the < RETURN > key, then force the loop variable to a terminating value and exit to the caller.

**678Ø** Otherwise, just clear the screen and return to the caller.

**68ØØ** Subroutine to mark fields for special processing. Advise user how to proceed.

**681Ø** Remind user how to terminate the input list.

**682Ø** Limit inputs to 2-digit integer values. Accept numeric input. If value outside range of valid field numbers, then issue error message, erase input from the display and try again.

**683Ø** Terminate this subroutine when user inputs 'Ø'.

**684Ø** Set the current Field Status array element to a '1' to serve as a marker. (Remember, array element is one less than field specified.) Then erase the current input line from the display and jump back to get another input.

**69ØØ** Subroutine to initialize the Field Status array elements to zero. Form loop, zero each element, exit to caller.

**691Ø** Message subroutine.

**692Ø** Message subroutine.

**693Ø** Message subroutine.

**694Ø** Message subroutine.

**7ØØØ** FIND ROUTINE. See if file is empty. If so, clear the flag and return to the Secondary Menu.

**71ØØ** Clear the screen. Display function header. Call subroutine to display the last record number in the file.

**711Ø** Call subroutine to get record numbers to process. If flag is set upon return, then clear flag and abort operation by returning to the Secondary Menu.

**72ØØ** Call subroutine to see what fields, if any, are to be suppressed from the output. This subroutine also finds out if the user wants to output record numbers and whether output is to the printer or screen.

**721Ø** Clear the screen. Ask user for the name of the field to be searched.

**722Ø** Limit response to 1Ø characters. Accept alphanumeric name.

**723Ø** Call subroutine that checks validity of a field name. If flag is set upon return, then could not find field name. Clear the flag and jump back to try again.

**73ØØ** Provide a few blank lines. Ask for the search string.

**731Ø** Limit length of search string to maximum length of the field being searched. Accept alphanumeric input. Provide a few more blank lines.

**732Ø** Initialize Character Position counter. If not processing the first field in the record, then loop to set up the Character Position counter to the start of the field that will be searched.

**733Ø** Clear the screen. If Printer flag is set, then issue command (in DOS format) to turn on the external printer. Issue a few linefeeds.

**734Ø** Display operation header. Show records being scanned.

**735Ø** Also verify the search string. Provide few blank lines for separation. Then call the subroutine that displays the output format.

**74ØØ** Form a loop over the range of records being searched. (The array elements are one less than values specified.)

**741Ø** Extract the specified field from a record.

**742Ø** Form a loop to position the search string to all possible locations within the field.

**743Ø** See if can find a match with the search string. Jump over output section if no match.

**744Ø** Call subroutine to provide output on a match condition.

**745Ø** Then force the loop variable to terminate the searching of the current field.

**75ØØ** Loop to continue search in current field. Loop to search next record.

**751Ø** Terminate the routine with some linefeeds. Turn off the printer. Go back to the Secondary Menu.

**79ØØ** Subroutine to check the validity of a field name. Start by initializing Valid flag to a negative value. Form loop to scan the names of all fields defined for records in the file. Look for a name match. If found, force the loop to terminate and set the Valid flag value to indicate the Field Name array element where the match occurred.

**791Ø** If no match, continue scanning field names.

## Line-by-Line Commentary (continued)

**792Ø** If field name has been matched, then return to caller with the Valid flag pointing to the appropriate Field Name array element.

**793Ø** Otherwise, clear the display and tell the user that the field name given is not valid. Set a delay value. Call the delay routine so user has time to read message. Provide a few blank lines after the message.

**794Ø** Now display the valid field names for this file.

**795Ø** Form a loop to scan the Field Names array. Display each name. Set a delay value and provide some time between each name. Loop until have shown all names.

**796Ø** Now set the Error flag to notify the calling routine that the name given was not valid and exit to the caller.

**8ØØØ** SORT ROUTINE. See if the file is empty. If so, clear the flag and exit to the Secondary Menu.

**81ØØ** Clear the display, then present the function header. Call the subroutine to display highest record number in the file.

**811Ø** Call the subroutine that gets range of records to process. If flag set upon return, then clear it and abort this operation by returning to the Secondary Menu.

**812Ø** See if range specified covers at least two records. If not, then set up an error message and call subroutine to show the message. Exit to the Secondary Menu.

**82ØØ** Clear the display. Query user for the field to sort on.

**821Ø** Limit the input to 1Ø characters. Accept alphanumeric input.

**822Ø** Call subroutine to verify that a valid field name was entered. If flag set on return, name was not valid so clear the flag and jump back to try again.

**83ØØ** Clear the display. Throw up message that lets the user know that a sort is in progress. Indicate the range of the record numbers being ordered.

**84ØØ** Set starting and ending values to point to the records. (Remember, array elements are one less.) Initialize the Character Position counter. If not dealing with the first field in a record, then form loop to find the first character position in the specified field.

**841Ø** Calculate the number of records being sorted.

**842Ø** Take half that number (to the lowest integer value).

**843Ø** Calculate the remaining number of records.

**844Ø** Initialize the Swap flag.

**845Ø** Form a loop to process a portion of the records being ordered.

**846Ø** Form a pointer to the array elements holding the records.

**847Ø** If the key field is numeric, then jump ahead.

**848Ø** Compare the field in the current record against the same field in the 'other part' of the file. If current field is less than or equal in value (alphanumerically), then no swap is needed, so skip over 'swapping' instruction lines.

**8490** Skip over the lines for comparing numeric fields.

**8500** If field is numeric, convert strings to numeric values when doing the compare. Skip ahead if swap of records is not needed.

**8510** 'Swapping' instructions. Store the current record in a temporary buffer.

**8520** Put the record from the 'other part' of the file into the current record position.

**8530** Now place the contents of the temporary buffer into the vacated 'other part' of the file to finish the swap.

**8540** Set the Swap flag after performing a swap.

**8550** Loop to do the next compare.

**8560** If the Swap flag has been set during a sweep, then repeat the sweep.

**8570** If the Swap flag has not been set during a sweep, then go back to cut the partially sorted file in half. Repeat the process until the requested range of records is in order.

**8580** Exit to the Secondary Menu when the sort is finished.

**9000** TALLY ROUTINE. See if file is empty. If so, clear flag and go back to the Secondary Menu.

**9100** Clear display. Put up function header. Tell user last record number in the file.

**9110** Get the range of records to be processed. If flag set upon return, clear it and abort operation by going back to the Secondary Menu.

**9200** Clear display. Query user for name of field that is to be summed.

**9210** Limit response to 10 characters. Accept alphanumeric input.

**9220** See if the specified name is valid. If not, clear the flag and jump back to try it again.

**9230** Check to see that field specified is indeed numeric. If not, set up error message. Display the error message and exit to the Secondary Menu.

**9300** Initialize summation register. Initialize Character Position counter. If not dealing with the first field in the records, then form loop to determine the starting character position of the specified field.

**9310** Form loop over the range specified. (Remember the array elements are one less.)

**9320** Update the summation register with the value in the field.

**9330** Loop to sum over the range of records specified.

**9400** When finished summing, clear the screen and announce the results of the tally. Show the name of the field summed.

**9410** Show the range of record numbers included in the tally.

**9420** Display the tally.

**9430** Provide a few blank lines. Give user as much time as desired to view the results.

**9440** Go back to the Secondary Menu when any key is pressed.

**10000** ERASE FILE ROUTINE. Clear the display.

**10010** Verify that user's command is to erase the current file.

**10020** Limit response to a single character. Accept alphanumeric input.

## Line-by-Line Commentary (continued)

**10030** If response other than (Y)es, abort by returning to the Primary Menu.

**10040** If user confirms file is to be erased, then CLEAR all the variables. Note that the Defined flag will automatically be cleared at this point. Clear the display. Tell user that the job has been accomplished. Exit to the Primary Menu.

**11000** DISK CATALOG ROUTINE. Clear the display. Turn on error trap.

**11010** Use DOS directive to tell disk to display the catalog.

**11020** Let user view catalog as long as desired. Give directions for terminating the operation.

**11030** Exit on any key. Remove the error trap before leaving. Depart to the Primary Menu.

**12000** EXIT ROUTINE. Clear the display. Tell user consequences of leaving the data base program. Verify that this is OK.

**12010** Limit response to single character. If not (Y)es, then go back to the Primary Menu.

**12020** Otherwise, clear the screen and exit the Data Base Program.

**20000** READ DISK ROUTINE. If Defined flag is set, then see if user wants to discard file that is in memory. File must be clear before a new one is permitted to be read from the disk.

**20100** If Defined flag not set, then OK to process command. Clear the screen. Display the function header. Provide a few blank lines after the header.

**20110** Initialize all variables. Dimension arrays. Call upon subroutine to fetch the name of the file that is to be obtained from the disk. Set an error trap during I/O operations.

**20200** Open the specified disk file.

**20210** Set the disk read mode.

**20300** Input the elements of the Field Length/Field Status array.

**20310** Input the elements of the Field Names array.

**20320** Input the records of the file.

**20400** Close the specified disk file.

**20410** Set the file Defined flag. Remove the I/O error trap. Go back to the Primary Menu.

**20900** Subroutine to get file name for I/O operations from the user. Query user.

**20910** Limit response to maximum of 30 characters. Accept user's alpha-numeric input.

**20920** Form DOS control characters (<C/R>+<CTRL/D>) for passing commands to the disk unit. Return to caller.

**30000** WRITE DISK ROUTINE. See if a file is in memory. If not, clear the flag and go back to the Primary Menu.

**30010** Make sure present file has something in it. If not, clear the flag and go back to the Primary Menu.

**3Ø1ØØ** Clear the display. Show the function header, then a few blank lines.

**3Ø11Ø** Fetch a name for the file from the user. Set up an error trap during I/O operations.

**3Ø2ØØ** Open a file using the given name.

**3Ø21Ø** Now delete the file in case an updating is taking place.

**3Ø22Ø** Re-open the same file to re-establish the name.

**3Ø23Ø** Set the file write mode.

**3Ø3ØØ** Write the field parameters information on the diskette.

**3Ø31Ø** Write the field names.

**3Ø32Ø** Write the records in the file.

**3Ø4ØØ** Close the file.

**3Ø41Ø** Remove the I/O error trap. Exit to the Primary Menu.

**3Ø9ØØ** Subroutine to trap disk I/O errors. Only used if an error occurs during disk operations. Clear the screen. Tell user about the problem.

**3Ø91Ø** Wait for a key to be pressed. Clear the error trap. Go back to the Primary Menu.

**4ØØØØ** PRIMARY MENU ROUTINE. Clear the screen. Space down the screen a ways so as to center the menu.

**4ØØ1Ø** Display Primary Menu option number 1.

**4ØØ2Ø** Display Primary Menu option number 2.

**4ØØ3Ø** Display Primary Menu option number 3.

**4ØØ4Ø** Display Primary Menu option number 4.

**4ØØ5Ø** Display Primary Menu option number 5.

**4ØØ6Ø** Display Primary Menu option number 6.

**4ØØ8Ø** Display blank lines in place of unused option positions.

**4ØØ9Ø** Display Primary Menu option number 9.

**4Ø5ØØ** Place the cursor at the bottom of the screen. Wait for a key to be pressed. Strip the code inputted when a key is pressed by subtracting 48 (decimal) from the ASCII code. See if one of the digits Ø through 9 was inputted. Loop to this same line if not a digit key.

**4Ø51Ø** Go to the appropriate routine when a valid digit key is pressed. (Note that digits Ø, 7, and 8 cause the Primary Menu to be repeated as no functions have been assigned to those menu positions.)

**41ØØØ** SECONDARY MENU ROUTINE. See if there is a file in memory. If not, clear the flag and go to the Primary Menu.

**41Ø1Ø** Clear the screen. Space down to center the menu.

**41Ø2Ø** Secondary Menu option number 1.

**41Ø3Ø** Secondary Menu option number 2.

**41Ø4Ø** Secondary Menu option number 3.

**41Ø5Ø** Secondary Menu option number 4.

**41Ø6Ø** Secondary Menu option number 5.

**41Ø7Ø** Secondary Menu option number 6.

**41Ø8Ø** Secondary Menu option number 7.

**41Ø9Ø** Secondary Menu option number 8.

## Line-by-Line Commentary (continued)

**41100** Secondary Menu option number 9.

**41500** Place the cursor at the bottom of the screen. Wait for a key to be pressed. Subtract 48 from the ASCII code of the key. If the result is not 1 through 9 then stay on this line.

**41510** Otherwise, go to the appropriate routine. Note that digit Ø will cause the Secondary Menu to be repeated.

**41900** Subroutine to see if a file has been defined. If the 'Defined' flag is not set, then clear the screen and notify the user. Set up a delay value and call the delay subroutine to give operator time to read the message. Set a flag to notify the calling routine.

**41910** Return to the caller.

**50000** Subroutine to accept alphanumeric inputs. Set a maximum string length default value at this entry point.

**50010** Clear the input buffer.

**50020** Wait for a key to be pressed. Check for a backarrow (used to delete characters). Jump ahead if character is not a backarrow.

**50030** When have a backarrow and have more than one character in the input buffer, then chop off the last character in the buffer. Display the new cursor position and erase the rest of the input line from the screen. Go back to wait for another input.

**50040** If the input buffer only has one character in it when a backarrow arrives, then null the buffer, show the cursor position, and erase the input display line. Go back to get another character.

**50050** If fall through to this line, then the input buffer is already empty so ignore any attempt to backspace!

**50060** If character is a comma, load the comma error message. Display the error message. Throw away the comma. Go back for another input.

**50070** Jump ahead if the character is not the < RETURN > key.

**50080** On detecting a < RETURN >, see if there is something in the input buffer. If so, exit to the caller with the alphanumeric string in the input buffer (W$).

**50090** However, if the buffer is empty, then load an error message and call the subroutine to display it. Ignore the < RETURN > and go back for another character.

**50100** See if the new character will cause the buffer length to exceed the maximum allowed value. If so, set up an error message. Display the message. Discard the new character. Go back to look for a < RETURN > or backarrow!

**50110** If the character is valid and will not overflow the input buffer, append it to the buffer, echo it on the screen, go get the next one.

**51000** Subroutine to accept numeric inputs. Set maximum default length and clear the Integer flag at this entry point.

**51010** Initialize the input buffer.

**51Ø2Ø** Use the alphanumeric input routine to fetch a string.

**51Ø3Ø** Keep out leading null characters to avoid system problems. If find one, then handle it as an error.

**51Ø4Ø** First character may be a decimal point or the minus sign, provided there is more than 1 character in the buffer.

**51Ø5Ø** First character may be any decimal digit.

**51Ø6Ø** If reach here, set up an error message, display it to the user, erase the last input line from the display and go try again.

**51Ø7Ø** Check for leading/trailing zeros or the presence of any invalid characters in the string. Do this by saving the original length of the alphanumeric input buffer. Next, convert the string to a numeric value, then convert it back to the string format. The final length should match the original length, else consider it an error condition.

**51Ø8Ø** If the Integer flag is not set, can exit to the caller at this point with the numeric value and its equivalent in string format (in W and W$, respectively).

**51Ø9Ø** If the Integer flag is set, then need to check that the value is indeed integer. If not, set up an error message, display the message, erase the input line from the display and go try again for a legitimate integer value.

**511ØØ** If value is integer, exit to caller. Integer numeric value is in variable W, its string representation in W$.

**595ØØ** Error message. (Concerning attempted use of a comma.)

**5951Ø** Error message. (When maximum length has been reached.)

**5952Ø** Error message. (For a null input.)

**5953Ø** Error message. (When numeric input is not valid.)

**5954Ø** Error message. (Noninteger value when integer required.)

**5955Ø** Error message. (Improper format.)

**5956Ø** Error message. (Numeric value outside allowed range.)

**598ØØ** Subroutine to display an error message. Save the current horizontal and vertical cursor position. Move the cursor to the first column position near bottom of the screen.

**5981Ø** Call subroutine to give an audible warning.

**5982Ø** Display the error message near the bottom of the screen.

**5983Ø** Set delay value. Call delay routine to provide time for the user to assimilate the error message.

**5984Ø** Restore the cursor to the start of the error message line. Erase the error message from the screen. Now restore the cursor to its original position and exit to the caller.

**5985Ø** Subroutine to erase a line from the screen. Position the cursor to the start of the line. Erase the line. Exit to caller.

**5995Ø** Subroutine to give an audible alert. Issue a beep using the monitor. Delay a little bit. Issue another beep. Exit to the caller.

**5997Ø** Subroutine to provide a programmed delay. Variable I holds the delay value when this subroutine is entered. Form a loop and count down to zero. Exit to the caller when done.

## Line-by-Line Commentary (continued)

**59980** Subroutine to clear the screen and space down a few lines.
**59990** Subroutine to space down a few lines.

## Variables Usage

The following list shows the primary usages assigned to each variable name used in the data base management program.

**A$** —Temporary working buffer.
**A$(*)** —Field names.
**A(*)** —Field and file control array (usage explained in Chap. 5).
**AA$** —Used as a flag to indicate if a file has been defined.
**B** —Temporary working register (number of characters allowed in a field, starting position of a field).
**B$(*)** —Records array.
**B(*)** —Used for flagging fields (to suppress output, inhibit selection during Change Operations).
**C** —Temporary counter; printer-selected flag.
**CH** —Horizontal cursor position register.
**CV** —Vertical cursor position register.
**D** —Flag (suppress display of record numbers) and temporary register.
**E** —Character position counter.
**E$** —Temporary register used for error messages.
**F** —Temporary working register.
**G** —Error-condition flag. Used to notify the calling routine if a subroutine is aborted.
**I** —Temporary loop counter.
**I$** —Input character buffer.
**J** —Temporary loop counter.
**K** —Temporary loop counter.
**L** —Temporary loop counter.
**LC** —Maximum number of characters to be allowed during input.
**LD** —If nonzero, only integer values will be considered as valid by the numeric input routine.
**LI** —Temporary register.
**T$** —Temporary record working buffer.
**W** —Numeric input register.
**W$** —String input register.
**X** —Temporary register; alternate input register.
**X$** —Temporary string register; alternate string input register.
**Y** —Temporary register.
**Z** —Temporary register.

## Table of Variables

```
A$
2200 2800 3150 4140

A$(*)
1130 1900 2830 6630 6630 7340 7900 7950 9400 20310 30310

A(*)
1250 1280 1300 1310 1420 1420 1430 1440 1900 2100 2200 2220 2220
2230 2240 2250 2250 2800 2810 2850 2850 2870 2870 2900 2900 3130
3140 3170 3180 3200 3200 4130 4180 4200 4210 4220 5200 5200 5210
5210 5210 5720 5760 5800 5900 6610 6630 6640 6640 6720 6730 6740
6820 7310 7320 7410 7420 7450 7900 7900 7950 8400 8470 8480 8480
8500 8500 9230 9300 9320 20300 20320 30300 30320

AA$
1000 1450 20000 20410 41900

B
1240 1310 1310 1310 1430 1430 1430 1430 1430 1440 1440 2100 2240
2240 3150 3180 3180 4140 4180 4180 4210 4210 7310 7320 7320 7340
7410 7420 7450 7900 7900 7920 8400 8400 8470 8480 8480 8500 8500
9230 9300 9300 9320 9400

B$(*)
1900 2100 2220 2220 3140 3140 3150 3170 3170 4170 4170 4180 4180
4180 4190 4190 4190 5200 5200 5210 6730 7410 8480 8480 8500 8500
8510 8520 8520 8530 9320 20320 30320

B(*)
1900 4040 4150 6610 6720 6840 6900

C
1430 1430 6210 6570 6590 6750 7330

CH
59800 59840

CV
59800 59840

D
6540 6550 6700 8440 8540 8560 9300 9320 9320 9420

E
6720 6730 6740 6740

E$
1230 1240 8120 9230 59500 59510 59520 59530 59540 59550 59560
59820

F
5200 5200 5210 7320 7320 7320 7410 8400 8400 8400 8480 8480 8500
8500 9300 9300 9300 9320

G
2000 2000 2900 2900 3000 3000 3010 3010 3200 3200 4000 4000 4040
4050 4050 5000 5000 5110 5110 5700 5710 5750 5800 5800 6000 6000
6110 6110 7000 7000 7110 7110 7230 7230 7960 8000 8000 8110 8110
8220 8220 9000 9000 9110 9110 9220 9220 30000 30000 30010 30010
41000 41000 41900

I
1010 1450 2870 2870 2900 3010 3140 3140 3140 3140 4040 4040 4050
5200 5200 5200 5200 5210 5210 5800 5900 6230 6250 6710 6730 6770
6900 6900 6900 7320 7320 7320 7400 7410 7500 7930 7950 8400 8400
8400 8450 8460 8480 8500 8510 8520 8550 9300 9300 9300 9310 9320
```

```
9330 10040 20300 20300 20300 20310 20310 20310 20320 20320 20320
30300 30300 30300 30310 30310 30310 30320 30320 30320 40500 40500
40500 40510 41500 41500 41500 41510 41900 59830 59950 59950 59950
59950 59970 59970 59970 59970 59970

I$
40500 40500 41500 41500 50020 50020 50030 50040 50060 50070 50110
50110

J
7420 7430 7450 7500 8410 8420 8420 8430 8460 8570

K
6720 6720 6730 6740 6740 7900 7900 7900 7900 7910 7950 7950 7950
8430 8450

L
6610 6610 6620 6630 6630 6630 6630 6640 6640 6660 8460 8480 8500
8520 8530

LC
1120 1220 1270 1350 2270 2850 3120 3220 4120 4240 5140 5790 6510
6540 6570 6820 7220 7310 8210 9210 10020 12010 20910 50000 50100
51000

LD
1220 2850 3120 4120 5790 6820 51000 51080 51100

LI
51070 51070

T$
7410 7430 8510 8530 30400 30400

W
1230 1230 1240 1250 3120 3130 3130 3140 3150 3150 3170 3170 3230
3230 4120 5710 5720 5720 5740 5750 5760 5760 5770 6820 6820 6830
6840 51010 51070 51070 51090 51090

W$
1130 1280 1290 1360 1370 2220 2280 2870 2870 2870 2870 3170 3220
4180 4240 5140 6510 6540 6570 6580 6770 6770 7350 7350 7900 7930
9440 10030 11030 12010 20200 20210 20400 30200 30210 30220 30230
30400 30910 50010 50030 50030 50030 50030 50040 50040 50080 50100
50110 50110 51030 51040 51040 51040 51050 51050 51070 51070 51070
51070

X
1110 1130 1210 1250 1260 1280 1300 1310 1320 1360 1360 1420 1430
2100 2230 2240 2240 2240 2800 2810 2830 2830 2850 2850 2870 2870
3150 3170 3180 3180 3180 4140 4150 4180 4200 4210 4210 4210

X$
7350 7420 7430 7430 7450 20200 20210 20400 20920 30200 30210
30220 30230 30400

Y
5120 5200 5740 5760 6230 7340 7400 8120 8300 8400 8400 8410 8450
9310 9410

Z
4120 4120 4130 4130 4140 4170 4170 4180 4180 4180 4190 4190 4190
4220 4250 4250 5120 5200 5200 5200 5770 6230 6770 7340 7400 8120
8300 8400 8400 8410 8430 9310 9410

END OF VAR. LIST
```

## Table of Line Number Cross-References

```
  1000
40510

  1030
1000

  1100
1360

  1220
1230 1240

  1270
1290

  1310
1280

  1350
1370

  1400
1310 1320

  1900
1030 20110

  2000
2280 41510

  2200
2240

  2250
2230

  2800
2210 3160 4160

  2830
2810

  2870
2850

  2900
2000 3000 3200

  3000
41510

  3120
3130

  3140
3230

  3160
3180

  3200
3170
```

```
  3900
3100 3160

  4000
41510

  4100
4050

  4120
4130

  4140
4250

  4150
4210

  4200
4150

  4220
4200

  4900
4010 4100 4160

  5000
41510

  5100
5140

  5700
5110 6110 7110 8110 9110

  5710
5730

  5740
5720

  5750
5760

  5790
5710 5750

  5800
3010 4000 5000 6000 7000 8000 9000 30010

  5900
3100 4100 5100 5210 6100 7100 8100 9100

  6000
41510

  6500
6200 7200

  6530
6510

  6560
6540

  6570
6580
```

```
  6600
6220 7350

  6660
6610 6640

  6700
6240 7440

  6720
6700

  6740
6720

  6800
4030 6520

  6820
6820 6840

  6900
4010 6500

  6910
6500

  6920
6520

  6930
6530

  6940
6560

  7000
41510

  7210
7230

  7500
7430

  7900
7230 8220 9220

  8000
41510

  8200
8220

  8420
8570

  8440
8560

  8500
8470

  8510
8490

  8550
8480 8500
```

```
  9000
41510

  9200
9220

 10000
20000 40510

 11000
40510

 12000
40510

 20000
40510

 20900
20110 30110

 30000
40510

 30900
11000 20110 30110

 40000
1 1020 10030 10040 11030 12010 20410 30000 30010 30410 30910 41000
41510

 40500
40500 40510 40510 40510

 41000
1450 2000 2290 3000 3010 3120 3200 3220 4000 4060 4120 4220 4240
5000 5110 5210 6000 6110 6260 7000 7110 7510 8000 8110 8120 8580
9000 9110 9230 9440 40510

 41500
41500 41510

 41900
30000 41000

 50010
1120 1270 1350 2270 2860 3220 4240 5140 6510 6540 6570 7220 7310
8210 9210 10020 12010 20910 51020

 50020
50030 50050 50060 50090 50100 50110

 50060
50020

 50100
50070

 51010
1220 2850 3120 4120 5790 6820 51060 51090

 51060
51030 51070

 51070
51040

 59500
50060
```

```
 59510
50100

 59520
50090

 59530
51060

 59540
51090

 59550
1290 1370 6580

 59560
3130 4130 5730 5760 6820

 59800
1230 1240 1290 1370 3130 4130 5730 5760 6580 6820 8120 9230 50060
50090 50100 51060 51090

 59850
1230 1240 1290 1370 3130 4130 5730 5760 6580 6820 51060 51090

 59950
59810

 59970
1010 1450 2900 3010 5800 5900 7930 7950 10040 41900 59830

 59980
1010 1100 1330 1400 2200 2900 3900 4900 5100 5120 5800 6100 6210
6780 6910 6920 6930 6940 7100 7210 7330 7930 8100 8200 8300 9100
9200 9400 10000 10040 11000 12000 12020 20100 30100 30900 40000
41010 41900

 59990
1010 1200 1250 1410 2200 2260 2840 3010 3100 3160 3210 4020 4100
4160 4230 5130 5700 5740 5900 6210 6260 6260 6760 7300 7310 7330
7350 7510 7510 7930 9410 9430 11020 12000 20100 30100 30900 40000
40000 41010 41010

END OF LN# LIST
```

# 7 BELLS AND WHISTLES

Breathes there a soul that can resist adding to or making alterations to a program? Hardly. Thus, the latter part of this book has been designed to actually help, rather than hinder, those venturesome comrades who feel the need to give in to their creative programming urges. The previous few chapters have provided basic technical information about the program. This final section gives ideas and encouragement to those who want to join the ranks of membership in the fraternity of "custom programmers."

## The First Rule

Never start monkeying around with a program until you have saved a few archive copies of the untouched original! Yes, you are authorized to keep multiple copies of the material provided in this publication for your own personal use. Since the publisher and I are being so open in providing detailed information about how this program works, we hope you will help us by telling your friends to buy their own copies. Doing so can encourage us to produce more material of this nature in the future.

It is also a good idea to periodically back up whatever version of the program you might be working on. After all, you never can tell when there is going to be a power failure or some kind of computer glitch. Better safe than sorry. Furthermore, getting in the habit of doing this can save a lot of grief if the development or modification efforts don't go exactly according to plan. After all, once you start modifying a program, it is possible to mess the thing up. Having a chronological series of copies of the program as work proceeds allows you to go back a few steps, if you suddenly discover your alterations no longer work as planned!

## Basic Customizing

Let's start with fairly simple and straightforward changes. One of the first possibilities in this category might be making room for more data.

There may come a time when you are working on an application that you realize you need "just a little more room" for your data file. What can you do to remedy the situation? (This treatise assumes that you need more memory in which to store the file, not that you need to store more than 999 records. That would be a different matter!)

You could remove portions of the data base program that you may not need as much as you require that extra space! For example, you might decide you could live without having the INSERT and CHANGE options in a particular application. Removing the portions of the program that provide those capabilities would leave more room in the memory of your system for your data file. You might consider, say, also getting along without the FIND and TALLY options. That would open up still more room.

Because of the modular way in which this program was constructed, it is quite easy to remove sections of the program. However, you must use a little consideration and "look before you leap." You should always use the table of line number cross references (provided in the previous chapter) before proceeding to take out program lines.

For instance, suppose you want to eliminate the INSERT option. You know (from reading previous chapters, I hope!) that the main statement lines for this operation begin at program line 3000. In-line operations are in the lower-numbered section of the 3000 block. Subroutines primarily associated with this operation would be in the higher-numbered section of the block. (For instance, at line 3900.)

### Use the Programmer Aids

A check of the table of line number cross references beginning at line 3000 immediately brings up an important consideration. That line is referenced by line 41510. That turns out to be the Secondary Menu routine. You now know that before you kick line 3000 out of your new version of the program, you better take care of what happens when you try to select the INSERT option from the Secondary Menu! One quick and dirty solution to this problem would be to simply alter line 3000 to read: GOTO 41000. That is, return control right back to the menu.

Of course, that method leaves something to be desired. For one thing, after you remove the ability to perform an insert, the Secondary Menu would continue offering a nonexistent option. A much more professional approach would be to modify the Secondary Menu so that it no longer served up the INSERT option. A little study of the program listing and commentary should give you insight into making the appropriate modifications there, if desired.

Let's return to the question of what can be removed from the section of the program (starting at line 3000) that originally provided the INSERT capability: Further examination of the program and line number cross reference table reveals that, with the exception of line

3ØØØ, which I have already discussed, all of the lines in the 3ØØØ block (3ØØØ-39ØØ) could be removed. This is because none of the other lines are referenced by routines outside of this block!

The same kind of situation exists for the CHANGE option. The first line of the module (4ØØØ) is referenced by the Secondary Menu. The rest of the lines in the 4ØØØ block are effectively "self-contained." Thus, they could be removed without affecting the rest of the program.

But be careful when it comes to knocking out the FIND option! A check of the line number cross reference table shows that the FIND operation contains a subroutine, starting at line 79ØØ. That subroutine is called upon by several "outside" routines. Removing that subroutine would cause those "outside" functions to "crash." So, you might wisely restrict your line deletions in the 7ØØØ block to the range 7ØØØ-751Ø. (Assuming, of course, that you make provisions for the Secondary Menu reference to line 7ØØØ.)

It turns out that you can safely bump off all of the TALLY option. That is, provided you take care of the initial entry point (line 9ØØØ) referred to by the Secondary Menu. By now I am sure you see the point. Don't just blindly wade into the program and lop off lines. Check to make sure anything removed is not needed by another section of the program that you still want to be able to use. Refer to the table of line number cross references to avoid problems.

Of course, those are not the only sections of the program eligible for elimination. They are mentioned as possibly being the most likely. Nor is the need for additional memory in which to store a larger data base the only reason for removing some options. I'll comment on that shortly.

### Decide First!

At this point, it is necessary to mention that you must make the decision to remove portions of the program (thereby freeing up additional memory for data) *before* you establish the data base.

Why? Because, at the conclusion of defining the format for a data base, the program (see lines 143Ø and 144Ø) determines how much memory is available for data storage. It then sets array element A(82) to the maximum number of records that can be contained in this space. If you use the original program to format a data base, it will limit the number of records (assuming it is less than the maximum 999 permitted) based on available memory. If you save that data base on a diskette, it will save that maximum allowable record number in array element A(82). Attempting to use the data base later, with a data base program that has been reduced in size, won't do a bit of good! A(82) will act to limit the data base to its original maximum size.

So, if you know that the size of a data base is going to be a problem, then do your homework first! Create a reduced-size data base manage-

ment program by removing unwanted options *before* actually defining a file. Got it?

(There are ways, of course, for advanced programmers to get around the problem of having an earlier defined program limited in size by the value in element A(82). I assume those with such qualifications can proceed well enough on their own.)

### Restricting Access

Earlier I mentioned that lack of memory might not be the only reason for wanting to remove functions from the program. Another reason could be *because you do not want some operators to have access to those options.* Suppose, for instance, that you run a small business. Perhaps you decide to computerize a customer list. Later you find out that it would be beneficial to allow other employees of your organization to use the list. However, you do not want them to have the ability to add to or alter the list. Sound like a plausible situation?

So, OK, you provide a special stripped-down version of the data base program that is to be used by other people. In it, you remove the capability to APPEND, DELETE, INSERT, or CHANGE any data. You might want to limit other operations. I am sure you get the picture. Presto. You have a program you can give to other people. It will allow them to list the information, perhaps sort it on different keys, obtain tallies, and so forth. But, unless they are programmers, you have pretty much taken away the capability of altering your data base.

Of course, any time *you* want to, you can use the original version of the program to update or otherwise modify your data file(s).

### Going the Other Way

You can go in the other direction, too. That is, instead of taking away capabilities, you can add your own options. When you do this, though, you will reduce the amount of memory available for storage of a data base. (Unless you decide to swap unwanted routines for those of your own. This is often a practical alternative.)

The kinds of capabilities you might want to add are virtually limited only by the strength of your imagination and your own programming prowess.

However, the modular structure of this program, along with the fixed-field format chosen for its design, makes the implementation of many kinds of extra options quite straightforward.

There are plenty of convenient gaps in the line numbers assigned to the original program in which to place new routines (such as in the range 13ØØØ-19999).

The structure of the menu routines can readily be duplicated to provide whole screens full of new options. You can activate a new digit (such as Ø) in the original menus to provide a means of getting to a new sub-menu.

What kind of new capabilities might you want to provide? You will certainly know better than I, but here are some possibilities in the event you are just looking for excuses to dive in and enhance the package:

How about adding column-to-column mathematical operations? Perhaps an option named ADD would permit a user to specify the field names or numbers of two fields (columns) whose contents were to be added together. Maybe the result could be left in one of the original fields. Or, possibly you would permit the specification of a third field in which to store the result. Oh yes, you might as well let the user specify the range of record numbers over which this operation was to occur. (You want to provide plenty of flexibility, right?)

Once you had an option such as ADD functioning smoothly, you could use the basic structure you had developed to, perhaps, allow further operations such as SUBTRACT, MULTIPLY, and DIVIDE.

No need to stop there. You could provide other kinds of special mathematical operations. How about scanning all of the records in a particular field to find the maximum and minimum values, calculate an average, determine the mean or figure the standard deviation? The list could go on. . . .

You say you are not interested in creating mathematical capabilities? How about new ways to manipulate alphanumeric information? The ability to, for example, find all occurrences of a particular word or phrase in a field (for any or all records) and replace it with different data.

A nice capability to add to the program, if you plan to maintain extensive mailing lists, is this: The ability to search through a file, looking at the contents of a specified field, and then remove (delete) those records that contain (or do not contain) a specific key. That is, to essentially automate the FIND and DELETE operations that are in the program!

How about extending the program's I/O capabilities? Could you use a "file merge" feature? This, for example, might allow you to combine several small files. What about providing the ability to split a large file into two sections?

All of these types of operations could be added by skillful and resourceful programmers. While the actual description of such alterations are beyond the scope of this text, those ideas may provide food for thought . . . .

### Modifying Existing Routines

It is not necessary to create whole new capabilities, however, in order to enhance the operation of the basic program. Many frills and new

features can be added by modifying, often just slightly, the various routines in the package.

Suppose, for instance, that you are not happy with the fact that the original package does not allow the use of a comma. This design decision was made, primarily, to avoid potential I/O problems. But, you say you *must* have commas. OK, let's talk about how you might go about altering the program to allow their use. (At some risk, as will be pointed out shortly.)

Commas are kept out of a data base file by the operation of the alphanumeric input routine that begins at line 5ØØØØ. You could allow commas to be entered by deleting line 5ØØ6Ø. That line performs the comma screening procedure. But wait! A check of the line number cross reference table reveals that line 5ØØ6Ø is referenced by line 5ØØ2Ø. Examination of line 5ØØ2Ø indicates that removing line 5ØØ6Ø means you would have to change the last part of line 5ØØ2Ø. It would need to become "THEN 5ØØ7Ø" instead of the original "THEN 5ØØ6Ø."

With line 5ØØ6Ø taken out of the program, you could also eliminate line 595ØØ. That is the error message subroutine referred to by line 5ØØ6Ø. (You may, however, want to keep it handy, for a reason that will be pointed out shortly!)

Once those changes had been made, you would find that you could indeed enter commas when using the data base management program. In fact, if you never had to save data files on diskettes, you might never detect any difference in the program's operation. (Other than that you now could input commas!)

Alas, however, all is not necessarily well. You would quickly discover this fact if you attempted to write records containing commas (by saving the data file) to a diskette and then read the file back into memory. Some of your data would be missing. Chances are that the read operation itself would, as the saying goes, "bomb." You see, commas that are not enclosed within quotes cause the DOS to interpret inputs differently than when they are inside quoted strings.

But all is not lost. It is possible to further modify the program so that it can write and recover data files to and from a diskette even when there are commas present *in the fields making up the data records.* All that is needed is to alter line 3Ø32Ø to read:

```
FOR I=Ø TO A(81):T$=CHR$(34)+B$(I)+CHR$(34):PRINT T$:NEXT I
```

What this does is throw quotation marks around each record as it is written to the diskette. No changes have to be made to the diskette read routine. This is *because the DOS will automatically strip off those quotation marks* and ignore any commas buried within the quoted strings!

Think you have the situation licked? Well, there are still a few pits left to fall into. What happens, now that commas are allowed during *any* user input, if someone puts a few commas in the strings used to name fields (i.e., in elements of the Field Name Array, A$)? Trouble will happen, friends, if you try to recover *that* file from a diskette.

Are there ways around this situation? Sure:

1. Tell users *never* to use a comma in the name of a field. (This approach is not recommended. Users never believe "never.")
2. Further modify the disk write routine to take care of this potential situation by changing line 3Ø31Ø to:

```
FOR I=Ø TO 39:A$=CHR$(34)+A$(I)+CHR$(34):PRINT A$:NEXT I
```

(You did check the table of variables to make sure you could use the string variable A$ in this situation, didn't you?)

3. Create a separate input routine for use when inputting field names *that did not allow the inputting of commas.*

OK, you are the boss now; it is *your* program. Which do you think is the best method? Do you think we have thought of all of the possible ramifications of allowing the use of commas within the program? Well, as I said, you are now creating your own customized version, so you make the choices you feel are best for your own particular circumstances.

Perhaps, though, after this little discussion, you will appreciate why I did not allow the use of commas in the published version. It was because *that was the safest alternative* when considering that totally novice users would likely be attempting to use the package. You, the venturesome reader, can now make the choice as to whether you want to trade off program integrity or "safeness" for the benefits you may derive from being able to use commas. Be careful not to chortle to yourself too loudly as you enjoy whatever new comma-commanding benefits you decide to install. There may still be pitfalls lurking to catch the unwary!

### From Numbers to Names

You may have noted, when using the data base program, that some options direct the user to identify fields by name. Other options call for identification of the field by number.

Why the two methods? Which method is better? Well, there are valid reasons for using both (as was done). But, there can be little argument that which method to use is largely a matter of personal preference.

I found, when designing the program, that for me it seemed most comfortable to be able to specify a field by *name* when I wanted to search through records. This was also the case when designating which field to sort or tally on.

On the other hand, I found there were places where having to type in the names of a lot of fields were cumbersome. One such place occurs when you are specifying which fields to suppress when using the LIST or FIND options. There it seemed more convenient to just enter the numbers of all the fields that I did not want to have outputted. Experience provided further impetus for staying with this method of selection. I have found that when I have been working long enough with a particular application to know which fields are not needed during a listing operation, I know what positions they occupy within a record. Field position corresponds to field number. Furthermore, by the time I get around to being concerned with doing selective listings with various fields being suppressed, I have plenty of help at hand. That help is in the form of previously made listings whereby *all* the fields are shown. The header of such listings contains the field numbers of all the fields *along with their names.* Thus, it is an easy matter to read down such a header and merely enter the numbers of the fields that are to be suppressed.

But suppose you are not impressed by that line of reasoning. You want to be able to enter the *names* of the fields that are to be suppressed instead of their numbers. Well, as I have said before, now it is *your* program and you are free to change it to suit your tastes. It really is not too difficult to implement the kind of variation being discussed here. Changing the program by adding the lines shown here would be one way to accomplish the objective:

```
6200 GOSUB 6900

6202 GOSUB 59980:GOSUB 6910

6204 LC=1:GOSUB 50010:IF W$<>"Y" THEN 6210

6206 GOSUB 59980:PRINT "SUPPRESS FIELD NAMED?":PRINT

6208 LC=10:GOSUB 50010:GOSUB 7900:IF B=>0 THEN B(B)=1

6209 GOTO 6202
```

A line-by-line commentary of the new lines will help clarify the matter:

**6200** Call subroutine that initializes Field Status array to zero.

**6202** Clear the display. Call message subroutine that asks if user wants to suppress a field.

**6204** Limit response to one character. Fetch response from user. If response is not (Y)es, then skip the rest of this procedure.

**6206** Clear the display. Query user for name of a field to be suppressed.

**6208** Limit field name response to maximum of 10 characters. Get the user's input. Call subroutine to check the field by that name exists. If B is not

negative upon return, then use value of B as a pointer to the Field Status array element that is to be set, indicating that field is to be suppressed.

**6209** Loop back to see if any more fields are to be suppressed.

You can see that this method calls upon various routines that are already present in the program to accomplish the objective. (You might, referring to line 6202, want to substitute a slightly different message than that provided by calling the message subroutine at line 6910.) The point being made here is that such essentially cosmetic changes can readily be made at your discretion. All it takes is a little studying of the listing and supporting documentation to locate the operations you want to duplicate. Then, you judiciously blend the portions you want into the appropriate place(s) in the overall program.

You could, for instance, perform a *similar* alteration to the FIND option, so that you could again use the names of fields that are to be suppressed. A slightly different variation of this scheme could also be applied to the CHANGE option. There, you could apply the new procedure to enable a user to specify, by field name, those fields that were to be subject to change.

### From Names to Numbers

Of course, you might be interested in going the other way. That is, you might want to alter, say, the FIND routine so that you could specify the search field by number, instead of name. Here is one way that could be arranged:

```
7210 GOSUB 59980:PRINT "SEARCH FIELD NUMBER?":PRINT

7220 LC=2:LD=1:GOSUB 51010:B=W-1

7230 IF B<0 OR B=>A(80) THEN GOSUB 59560:GOSUB 59800:GOSUB 59850:
     GOTO 7210
```

The commentary for these replacement lines would be as follows:

**7210** Clear the display. Query user for field number that is to be searched.

**7220** Limit number of digits to just two. Number inputted must be an integer. Fetch the user's response. Subtract 1 from user's input as array elements start at zero, not one.

**7230** Check that field number specified is valid for the current data file. If not, display error message, then go back to try again. Otherwise, continue program. Variable B holds pointer to field selected.

The same instructions could be applied in the SORT option (at lines 8200, 8210, and 8220) and the TALLY operation (lines 9200, 9210, and 9220). You would, of course, want to alter the user query (in line 7210 of

the illustrative routine) to correspond with the type of operation being performed.

## Changing CHANGE

The CHANGE option was designed with the assumption that a person making changes to a data base would be working from a list (produced by the program) that was being updated. This might be the case, for instance, when someone was updating a mailing list. Typically, an archival list would be marked up with the necessary changes. This would be in front of the computer operator entering the data for reference.

However, when reviewing the program, an associate pointed out that there could be applications where it might be more convenient not to have to refer to a previously generated printed list. One such case might be where a field contained information regarding, say, the status of a project. Periodically that status might need to be changed to reflect progress (or the lack of such).

It would be nice, in such a situation, if the CHANGE routine would display the current contents of each field that was subject to alteration. This can be arranged with relatively little effort. As a matter of fact, just adding a single line and altering one other can provide such capability:

```
2845 IF G=1 THEN PRINT MID$(B$(Z-1),B,A(X)):FOR I-1 TO A(X):PRINT
     "-";:NEXT I:PRINT

4160 GOSUB 4900:GOSUB 59990:G=1:GOSUB 2800:G=0
```

Line 2845 is entirely new. Line 416Ø is a modification of the original. These lines could be commented as follows:

**2845** If flag (G) is set, display the current contents of the field being accessed by the CHANGE option. Draw a line of dashes underneath. Length of the dashed line to be equal to the number of characters permitted in the field.

**416Ø** Call subroutine to display CHANGE function header. Then provide a few blank lines. Set a Display flag to enable data input subroutine to display current field contents. Call the subroutine that accepts data inputs. Clear the Display flag upon return.

With these modifications, you would be able to see the current contents of a field whenever you made changes. You can then elect to input different data or copy the current contents of the field.

You might be inclined to proceed further. It might be quite nice to be able to elect to keep the present data in a field after it had been displayed, rather than changing it! But, providing this feature is not a trivial task. If you want to approach the project, I'll give you a few tips.

First, you will need to provide a new type of character input routine. One that will accept a "null" input (that is, just the <ENTER> key). The current version blocks out inputs that do not contain at least some other valid character.

Second, you will need to differentiate between a null input and a "data" input. The former could be used to indicate that *the current contents of the field were to be left unchanged.*

Third, watch out for the fact that you can be dealing with both "types" of fields: alphanumeric and numeric! How are you going to handle numeric inputs?

Can you find ways to modify the original input routines (starting at lines 5ØØ1Ø and 51Ø1Ø) so that they could provide the kind of operation necessary to provide this new capability?

Remember to use all the programming tools provided: the program listing, commentary, variables, and line number cross referencing documents. They can be invaluable when facing such a challenge.

If you are looking for still more to do with the CHANGE routine, how about these possibilities: (1) modify the setup procedure so that the user can specify those fields that will *not* be subject to change within each record (be careful, this can be a little trickier than you might think at first glance); or (2) if you implement the capability for the user to review the current contents and opt to leave it unchanged, then you could simply allow all fields in a record to be processed. Providing for this should be a "piece of cake," if you have the savvy to accomplish the former part!

### Fixing Up FIND

The possibilities for enhancing the FIND option are many. You might want to go to work making your own really high-powered DBM by adding to this operation. What are some things you might do to provide more search power? How about these:

You could essentially duplicate and "nest" the basic search operation *so that it examined two or more fields within each record.* Furthermore, you might consider giving the user options to perform Boolean logic combinations among those fields. That is, to specify search strings within each field. Then to further specify that a record is to be outputted only under selected logic criteria, such as if a match is found in field A *and* B or if it is found in field A *or* B, etc.

You might also consider permitting multiple search strings to be specified within a single field. Again, this could be further amplified to allow Boolean logic selections among those intrafield strings!

How about adding more search string versatility? You might want to study the feasibility of providing for "wild-card" character positions within a search string. Or how about adding an option to look for characters according to position within a field. Thus, while the original search algorithm will, for instance, consider it a match if a given string is

found *anywhere* within a field, you might want to be able to consider it a match only if it occurs in a specified position.

A word of caution, however: Search algorithms can eat up a lot of memory. While large and powerful data base management programs you may have heard about often establish a reputation for their search capability, they do so at the expense of lots of memory. Normally those types of programs use disk overlays to provide a great many options. This method reduces the amount of memory utilized by the program at any given time. This is because only the section of the program currently in use (and a few supporting utility subroutines) are in memory at any one time.

This relatively small, memory-resident data base management program is not designed to facilitate program "chaining" and overlays. So, don't get carried away with providing all kinds of search capabilities, only to realize you don't have room left for a substantial data base on which to exercise all that power! (But you can, of course, set up a whole library of modestly sized versions of this program. Each of which might offer differently tailored options!)

### Don't Forget This

Data bases established using this program are memory sensitive. This was explained earlier. It bears repeating its ramifications, however, especially if you wish to customize several versions with differing capabilities.

The essence of the matter is this: If you establish a data base in a system that has, say, 18K of memory available after the program itself is loaded, that is the amount of memory that the data base will be formatted to utilize. If you modify the data base management program by adding capabilities so that it expands in size, you may not be able to access data bases created by a smaller version.

Thus, the rule of thumb, if you intend to work with several different sized versions of the program, is this: Always establish data base files with the *largest* data base program (which will give you the *smallest* data base file). This will insure that the data base file will fit in memory regardless of which version of the program you are using in the future. Got it? (If not, think about it until you do!)

### Streamlining

Once you have used the program for awhile, you may find that certain options "bug" you. Perhaps, for instance, you *always* like to have record numbers displayed whenever you get a listing (or you *never* want them shown). Whichever the case, it will take just a few seconds for you to modify the program so that it simply invokes the method you prefer. Do

it! It will save a few seconds of time whenever you use the program and make the program more enjoyable for *you* to use.

You can employ this customizing concept at any "option" point in the program. If you never need to choose between certain alternatives, then fix the program up so that the selection process is eliminated at that point.

### This and That

Advanced programmers may want to "play around" with other kinds of modifications or enhancements.

One of the "quirks" of the original Apple II system is that it periodically has to "clean up" its character strings storage area. You probably will not notice this characteristic until you put a lot of data into a file. However, as a file becomes filled, this phenomenon presents itself. You may then begin to notice that, seemingly at random, the program seems to stop functioning for a few seconds.

This "few seconds" delay can gradually increase, as more and more data is added to a file, until it becomes substantial. It appears that the amount of time required by the Apple to "clean up" its strings storage area increases as the number of strings being manipulated expands and the amount of "free" memory left contracts. Naturally, when you are close to filling up a data base file, this condition exists in its worst form.

There are several solutions to this potential problem. One is to effectively "avoid" it by planning on not filling files to their near maximum capacity. My own experience has indicated that I seldom come close to filling up most of the data files I establish with this program. If I believe, at the start of a project, that a single file may be insufficient for a task, I generally opt to set up two files and split the data into logical portions. (Or else I consider using a much more powerful DBM package that stores individual records directly on a diskette.)

A second solution, which I have heard about but have not personally implemented, involves adding special routines to enhance the Apple's string-handling capabilities. Some readers who are programming "heavies" may be aware of the availability of such routines. If so, you may want to consider adding them to this program.

Machine-language programmers should be able to have a lot of fun enhancing this program. One area that you might want to tackle, if you have the requisite skills, is the SORT routine. It is one part of the program that could be considerably enhanced by implementing it in machine language. The sorting time of several minutes for a few hundred data items could be reduced to a matter of seconds!

Another routine that would make a good target for improvement via machine-language routines would be the FIND function. Again, the big advantage would be the speed with which a data file could be scanned using machine-language techniques.

The subject of machine-language programming is well beyond the intended scope of this book. Suffice it to say, however, that with the complete structure and operation of this data base management program exposed and documented, one has a solid foundation from which to jump off and produce such enhancements.

### Toward a Disk-Based Version

A memory-resident data base management program, such as the one that has been described in this book, is a good way for anyone to "get their feet wet" on the subject. It has many practical, utilitarian applications. It is relatively simple. Thus, it is a good educational tool. This applies equally to aspects of learning how to use a data base management program as well as how one is constructed. The interested, enthusiastic program explorer can quickly grasp essential concepts. Then, if desired, continue on to enhance or elaborate on the basic design.

But any program that requires that the data it is operating on be entirely resident in memory, obviously, restricts the amount of data that can be conveniently processed. The next step up, in terms of data base management systems, is the so-called "disk-based" system. This refers to a system wherein the data base file resides on a mass memory disk. Each time a piece of information relating to the data base is accessed or manipulated, it is physically retrieved from the mass memory device.

These days, a typical 5¼-inch diskette can hold from 1ØØ,ØØØ to 4ØØ,ØØØ bytes of data (depending on the equipment and technology being utilized). Larger 8-inch diskettes are approaching the capacity of a million or more bytes. And new "hard disk" drives are capable of storing tens of millions of bytes of information. Thus, any disk system can store considerably more data than the typical computer system's main memory.

Organizing the data directly on a disk has advantages and disadvantages. Its major advantage is that a far greater number of items (records) can be stored and referenced as being held in one file. Its disadvantages have to do with such parameters as operating speed and disk-accessing complications. In a well-designed disk-based system, the user will be essentially oblivious to these problems. To the program designer, finding the solutions presents definite challenges. Some people have spent most of their lives studying the ramifications!

In closing this book, I am going to point out something of interest to those readers who may want to consider going on to design a disk-based data base management program. The fixed record length format selected for use in this program was chosen because it could be the basis for an advanced disk-based program

Be advised, however, that some of the problems that can arise when implementing such a program are not trivial. Should you want to go plowing ahead, however, here are a few possibilities to consider.

Basic file formatting information could be stored as a "header" file on the disk. This would typically contain the type of information stored in the A$( ) and A( ) arrays of this memory-resident program. Note that some elements (such as the number of the last record in the file) might have to be accessed and rewritten frequently.

Information in the "header" file could then be used to access individual records stored on the disk. With all records in a given file defined to be of equal length, random access disk access techniques could be used to deal with individual records. This would provide for relatively fast individual record retrieval.

Creating a program that would allow one to build up, make changes to, list, and even search through records in a disk-based file is relatively straightforward.

The problems start to arise when one gets into aspects such as deleting records or sorting large data bases. Does one compact all the records "above" the deletion point, as was done in the memory-resident file? That could take a long, long time if each record must be accessed from a disk and then rewritten in another location. And what happens when one wants to sort a data base containing 2Ø Ø,Ø Ø Ø entries? The memory-resident sorting algorithm used in this program is simply impractical when the individual records must be shuffled about on a disk.

Intrigued? Give the matter some thought. Read some more books. Try your hand at your own ideas. After all, five years ago there was no such thing as data base management on a personal computer. There is plenty of opportunity for the enthusiastic, ambitious personal computer user with a flair for programming to come up with new techniques.

6282-6

# DATA BASE MANAGEMENT FOR THE APPLE™

NAT WADSWORTH

Learn the basics of storing and organizing information on your Apple II Plus or IIe home computer. This clearly written book includes DATA BASE, a simple, functional, and cross-referenced data base management program written in Applesoft BASIC. Using DATA BASE as a reference, the author shows how data base management techniques apply to common chores. Keep mailing lists in order and print address labels, maintain household lists or inventories, organize appointments, manage checking accounts, put together a computerized tutor, and keep track of investment portfolios. And all with microcomputer power.

Complete line-by-line commentaries and variable indexes enable advanced programmers to tackle more ambitious data-management tasks or more specialized reports.

***Also by Nat Wadsworth...***

***GRAPHICS COOKBOOK FOR THE APPLE™***

Use your Apple to draw, paint, and even write in 16-color, full-screen, low-resolution graphics. The author explains the basics of computer graphics and supplies complete Applesoft BASIC programs that turn simple DATA statements into a host of graphic elements. The book also supplies a reference library of data needed to produce more than 100 different characters, shapes, borders, and backgrounds — everything from triangles to trees, from mosaic patterns to lakes and mountains, from sailboats to flying saucers. You can also produce billboard-size letters from A to Z. #6278-8, paper, 72 pages.

***Another book of interest...***

***6502 SOFTWARE GOURMET GUIDE & COOKBOOK***

ROBERT FINDLEY

The essential guide to machine-language programming for the 6502 microprocessor — the chip that runs the Apple II Plus, IIe, and III; the Atari 400, 800, and 1200 XL; the Commodore PET, VIC-20, and 64; and many others. The book opens the new world of machine-language programming to the experienced BASIC programmer. Programmers who know assembly language programming for other microprocessors will find in this book a ready introduction to the 6502. Programmers of every level will also find a wealth of tried-and-true assembly language subroutines for common computer chores: base conversions, input/output routines, multiple precision-floating-point arithmetic, decimal arithmetic, and super-efficient search-and-sort routines. #6277-X, paper, 204 pages.

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

0-8104-6282-6

WADSWORTH

# DATA BASE MANAGEMENT FOR THE APPLE™

HAYDEN 6282-6